Partnership for Advanced Computing in Europe

SUMMER OF HPC

summerofhpc.prace-ri.eu

A long hot summer is time for a break, right? Not necessarily!
PRACE Summer of HPC 2020 reports by participants are here.

HPC in the summer of 2020?

Leon Kos

Challenging times with no travel over the summer! Going all virtual with 50 participants and their mentors at 12 PRACE HPC sites working on 24 projects. No problem!

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, presentation and a video.

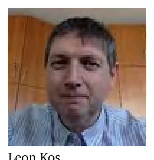
This year, training week was in held all virtual by Vienna Scientific Cluster (VSC) – PRACE partner from Austria! It was a great start to Summer of HPC and set us up to have an amazing summer! At the end of the summer videos were created and are available on Youtube as PRACE Summer of HPC 2020 presentations playlist. Together with the following articles interesting code and results are available. Dozens of blog posts were created as well. 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? HPC gave us exciting time and new outlook for a bright 2021 edition.

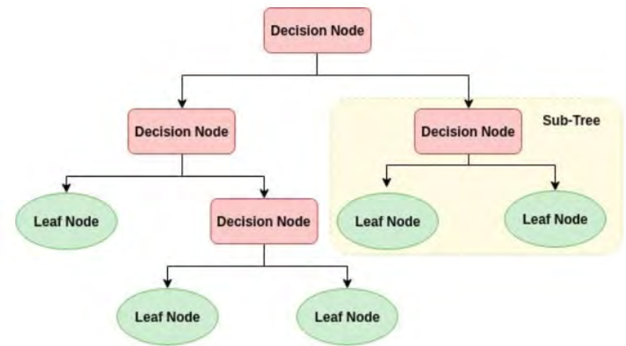
Contents

1 High Performance Machine Learning 3
2 Supernova Explosions Using HPC 6
3 When HPC meets integer programming 9
4 Anomaly Detection in High Performance Computing Systems 12
5 Persistent Memory checkpoint 15
6 ARM tooling 18
7 How to make Python code run faster 20
8 Breadth First Search 23
8.1 BFS Graph Traversal with CUDA 23
8.2 GPU Acceleration of BFS 25
9 Deep Neural Networks for galaxy orientation 27
10 Visualization of Molecules 30
11 High Performance Quantum Fields 32
12 FMM-GPU Melting Pot 35
13 Quantum Computing 38
14 Matrix exponentiation on GPUs 40
15 Predicting Job Run Times 43
16 Boosting Dissipative Particle Dynamics 46
17 Monitoring HPC Performance 49
18 TSM of HPC Job Queues 52
19 Accelerating Particle In Cell Codes 55
20 When HPC meets integer programming 58
21 Drifting in aSubmarine? Hold On! 61
22 Novel HPC Models 64
23 Hybrid Programming with MPI+X 67
24 Biomolecular Meshes 71

PRACE SoHPC2020 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

Improving the performance of Decision Tree CART algorithm (Machine learning algorithm) using MPI and GASPI Parallelization



High Performance Machine Learning

Machine learning is used to solve many complex real-world problems which are beyond the scope of human beings. ML needs big data, often in hundreds of terabyte, to make accurate predictions. Processing this data demands huge computational power and hence, becomes computationally expensive. The aim of this project is to improve the performance of a Decision Tree Machine Learning algorithm using High Performance Computing (HPC) parallelization tools such as Message Passing Library (MPI) and Global Address Space Programming Interface (GASPI). The results demonstrate that the parallelization of the algorithm significantly increased the data processing speed and efficiency of the algorithm.

The concept of Machine Learning (ML) has been around for a long time (think of the WWII Enigma Machine). The ability to automate the applications of complex mathematical calculations to big data processing has been gaining momentum over the last several years. Companies like google, IBM, Pinterest and Facebook etc are using ML to handle big data related problems.

ML is basically a method of data analysis providing a computer ability to automatically learn from data without being explicitly programmed. Once a ML algorithm is trained by a large enough input data set, it can interpret unseen data to make predictions.

Types of Machine Learning

ML methods are broadly classified into two main categories, Supervised and Unsupervised Learning. Each category contains many different methods and algorithms used to solve wide range of different problems. In supervised learning, labelled data is used to train the ML model. Firstly, the model is established by training the algorithm based on the previously labelled data, and then, the algorithm is used to make predictions on unseen data. The working principle of supervised classification is demonstrated in Figure 1.

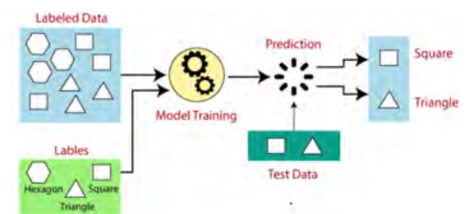


Figure 1: Machine Learning Based Classification

There are many different types of supervised learning algorithms such as Decision Tree, Polynomial Regression,

Logistic Regression Naïve Bayes, and K-Nearest Neighbors etc.

In Unsupervised learning, the data of interest is not labelled. The algorithm searches for previously undetected patterns in the data set with its self-organization. Figure 2 demonstrates the working principal of unsupervised learning.

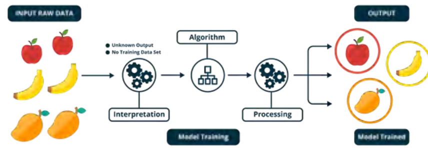


Figure 2: Example of how unsupervised learning works

Unsupervised Learning algorithms can be further split into various categories such as Partial Least Square, Fuzzy Means, K-Means Clustering, and Principal Component Clustering etc.

Big Data and HPC

Big data is a term used to describe extremely large volume of data which is impossible to be processed by using traditional methods. Even though it is hard to handle big data, today it is crucial component of predictive analytics.

HPC is the ability to perform complex calculations with high speed in parallel fashion. HPC tools like MPI, OpenMP or GASPI allow us to process big amount of data in highly efficient way.

Today, High-performance data analytics built using HPC tools is highly desirable. This is a result of highly growing demand for such tasks on supercomputer facilities.

Figure 3: A HPC facility

Decision Tree Classifier

Decision tree is a well-known supervised ML algorithm. It is a binary tree that predictions are made from the root (top of the tree) to the leaves(bottom of the tree). It can be used for both regression and classification problems. Decision Tree algorithms can be further classified as ID3, C4.5, C5.0 and CART algorithm. In this study, we implement CART algorithm.

In CART algorithm, each node is assigned with a GINI index which describes the purity of the node. The aim is to find optimal feature and threshold couples such that GINI index is minimized. The algorithm stops when the maximum depth is reached or when all the inputs are classified.

Gini index of n training samples split across k classes is defined as

$$G = 1 - \sum_{k=1}^{n} {p_k}^2$$

where p[k] is the fraction of samples belonging to class k.

Working Principal of Serial CART Algorithm

The program starts with a feature set and iterates through the sorted feature values for possible thresholds as shown in table 1.

Table 1: Features and classes of the data

	Feature 1	Feature 2	Feature 3	Feature 4	Feature 5	Feature 6	Class
0	-1.255	-1.332	36.848	-1.229	36.882	-1.323	1
1	-1.317	-1.255	36.782	-1.267	36.808	-1.266	1
2	-1.255	-1.277	36.823	-1.280	36.831	-1.323	0
3	-1.304	-1.317	36.830	-1.306	36.752	-1.300	0
4	-1.323	-1.255	36.782	-1.297	36.786	-1.300	0
...	...	...	...	...	...	...	...

During this process, it keeps track of the number of samples per class on the left and on the right. The same procedure is repeated for each feature set. The program then compares the Gini indexes to find the best couple for splitting. The splitting process recursively carries on at each node until the maximum depth is reached.

Larger training data set means that more info to learn for algorithm and this improves the prediction accuracy. But Large data means more calculations and hence it requites more computational power. Therefore, the ability to handle big data becomes critical. HPC tools can help in solving this problem.

MPI Based Parallellization Strategy

MPI is one of the most popular tools to develop parallel algorithms. It is a "standard" message-passing library designed for distributed memory systems. It provides specifications for creating separate processes on different computation units as well as establishing communication between each units.

Our parallelization strategy is distributing features among different computation units. In this approach, each processor core is responsible for one or more feature sets to survey all possible splits and compute corresponding Gini indexes to evaluate them. Therefore, in every recursion, each core finds a best feature and threshold couple minimizing the Gini index for their portion of the data. Next, all the best feature and threshold couples are gathered together at the master core 0. The master core compares them according to their Gini indexes and broadcast the best result to all the other processors.

Table 2: Feature set distributed according to the

	Core 0	Core 1	Core 2	Core 3	Core 4	Core 5	
	Feature 1	Feature 2	Feature 3	Feature 4	Feature 5	Feature 6	Class
0	-1.255	-1.332	36.848	-1.229	36.882	-1.323	1
1	-1.317	-1.255	36.782	-1.267	36.808	-1.266	1
2	-1.255	-1.277	36.823	-1.280	36.831	-1.323	0
3	-1.304	-1.317	36.830	-1.306	36.752	-1.300	0
4	-1.323	-1.255	36.782	-1.297	36.786	-1.300	0
...	...	...	...	...	...	...	...

Because synchronization is crucial for the calculations, communication between cores is established by standard synchronous send/receive commands. In addition, at the end of each recursion, master core sends the best result to all the other cores via broadcast command.

Results

The results shows that the higher the number of cores used, the higher the speed-up rates obtained. Figure 4 shows that total computation time is decreasing with increasing number of cores. Here, the computation time corresponds to run time of learning (data fitting) process of the algorithm.

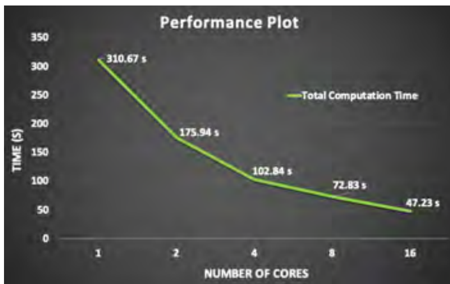


Figure 4: Performance plot of computation time

It is also seen that the rate of change is also decreasing with increasing number of cores. This is the result of inevitable communication overhead. In an ideal world, workers can do their job independently. But in real world, synchronization mostly is a must. There is no chance to escape from communication overhead required by synchronization in MPI parallelism. So in reality, speed up is always less than the increase of the number of workers.

Figure 5 shows the portions of the total computation time dedicated to calculation and communication. As it is seen that time spending for communication is increasing with increasing number of cores. Higher number of cores means that higher number of send and receive commands and for each command, cores should wait for each other. On the other hand, chunks of data which is transferred is getting smaller, and this makes faster the data transfer. Therefore, the increase in the communication time is not linearly increasing.

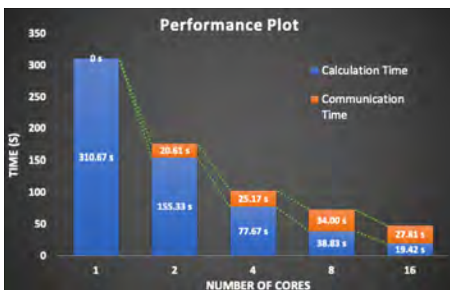


Figure 5: Performance plot of computation and communication time

How Is The Performance Further Improved?

As it is in the performance plots, time spending for synchronization is an important issue for a parallel program. In MPI's standard send/receive commands, cores blocks each other during the communication. This is called two-sided communication in which processors should wait each other until the message is successfully left by the sender and delivered to the receiver. We can't completely get rid of communication overhead, but we can overlap communication and computation task by using a different parallelization paradigm. This is shown in Figure 6.

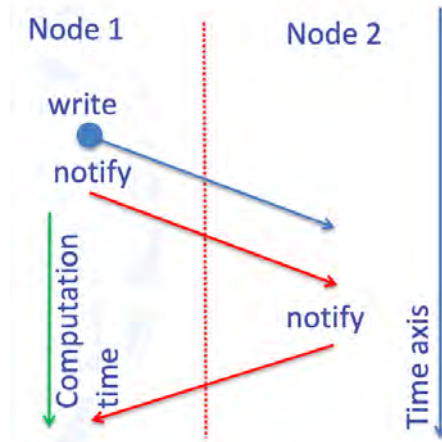


Figure 6: One-sided communication

An Alternative to MPI: GASPI

GASPI (Global Address Space Programming Interface) is considered as an alternative to MPI standard, aiming to provide a flexible environment for developing scalable, asynchronous and fault tolerant parallel applications. It provides one-sided remote direct memory access (RDMA) driven communication in a partitioned global address space. One-sided communication allows a process to make read and write operations on another processor's allocated memory space without the participation of other processors. Unlike two-sided communication, the processor whose memory is being accessed, continues its job without any interruption. This means that the processors continue their computations alongside with their communication. The working principle is demonstrated in Figure 7.

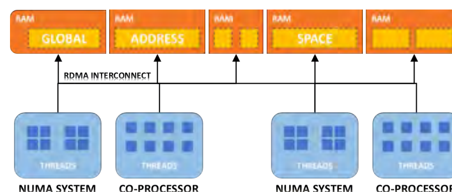


Figure 7: Working strategy of GASPI

GPI-2 is an open source programming interface allowing implementation of the GASPI standard. It is compatible with C and C++ languages to develop scalable parallel applications.

Conclusion

In this study, we focused on a popular supervised ML algorithm, Decision Tree Classifier, to investigate possible ways to improve its performance by using well-known parallelization tools.

As we have discussed that MPI parallelism has a great impact on the performance of the algorithm. On the other hand, the performance plot clearly demonstrates that communication overhead is the inevitable part of the communication. Performance can be further improved by using a different parallelism approach allowing asynchronous one-sided communication, such as GASPI/GPI-2 standard. Even though communication overhead is inevitable, it can be relieved by overlapping communication and computation.

References

[1] http://www.gaspi.de/gaspi/

[2] https://towardsdatascience.com/decision-tree-from-scratch-in-python-46e99dfea775.

[3] https://www.simplilearn.com/tutorials/machine-learning-tutorial/what-is-machine-learning

[4] https://medium.com/datadriveninvestor/tree-algorithms-id3-c4-5-c5-0-and-cart-413387342164.

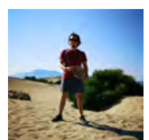
[5] Parallel and Distributed Computing lecture notes by M. Serdar Çelebi

[6] javatpoint.com/machine-learning-decision-tree-classification-algorithm

[7] https://vs.sav.sk/

PRACE SoHPC Project Title
High-Performance Machine Learning

PRACE SoHPC Site
Computing Centre of the Slovak Academy of Sciences, Slovakia



Cem Oran

PRACE SoHPC Authors
Cem Oran, Istanbul Technical University
Muhammad Omer, Manchester University

Muhammad Omer

PRACE SoHPC Mentor
Michal Pitoňàk, CCSAS, Slovakia

PRACE SoHPC Contact
Cem Oran, Turkey
E-mail: cemoran.01@gmail.com
Muhammad Omer, Pakistan
E-mail: eng.muhammad.omer@gmail.com

PRACE SoHPC Software applied
MPI

PRACE SoHPC More Information
www.gpi-site.com/gpi2
www.open-mpi.org

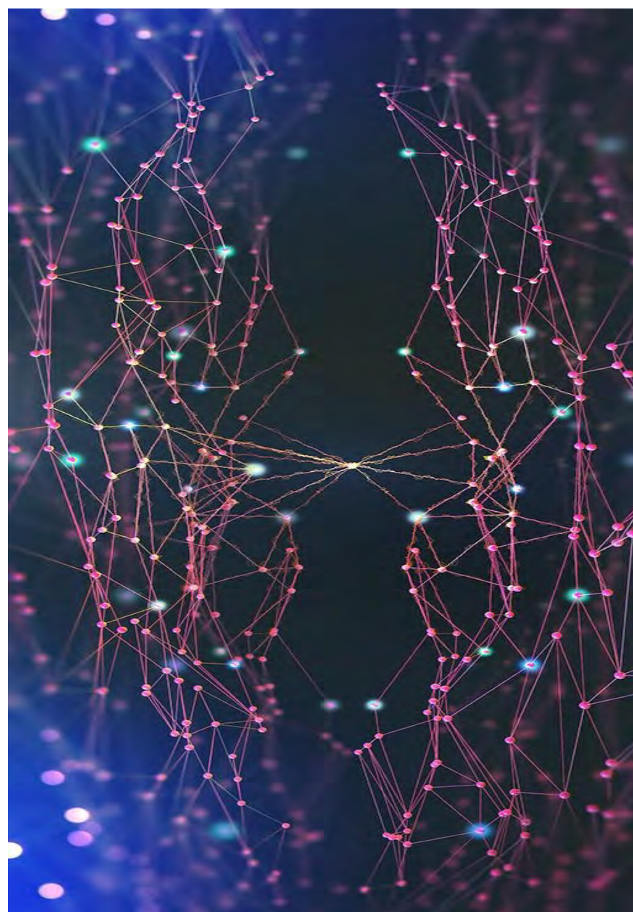
PRACE SoHPC Acknowledgement
We would like to thank our mentor Michal Pitoňàk for his time and effort.

PRACE SoHPC Project ID
2001

Implementation of Parallel Branch and Bound algorithm for combinatorial optimization

When HPC meets integer programming

İrem Naz Çoçan , Carlos Alejandro Munar Raimundo



Mixing mathematics and computers could get a very good results in efficiency. It could obtain the solution of a very large problem that for a human being could be unfathomable in a few minutes. And now, imagine that you can do it even faster.

Branch and bound method is one of the most used methods for solving problems based on decision making. In the particular case that concern us, is a discrete case, this means that you have to decide either A or B.

In order to understand this let us explain the max-cut problem. Imagine that you have a weighted graph and you want to divide the vertices into two subgroups. After that, the connection between two nodes that are in different subgroups is cut. The goal is to bipartition the vertices so that the sum of the weights on the cut edges is maximal.

Turning back to the branch and bound algorithm, it is organised by a rooted tree. All starts with root node, where upper bound and lower bound is set. In every step it has a node to compute, and by updating the bounds and checking them it is decided to branch to another two nodes or prune because you will not get any further going through that branch. An Figure 1 there is an example of this. With this, you have fewer options to explore, and it will take less time to obtain the optimal solution to this problem.

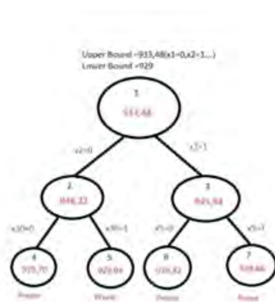


Figure 1: Diagram showing rooted tree of the branch and bound.

Notice that, we are working with max-cut problem that is a NP-complete problem, so we will need an heuristic to get an approximation to an optimal solution because getting the best solution could get even longer.

Nevertheless, we want thing going faster. As a consequence, this project came up. The aim of the project is to make exploration of the branch and bound tree even faster by concurrently exploring different branches. For that, we have done two different approaches: *master-worker approach*, and *one-side communication approach*.

In every modern computer, there is more than one core in a processor, so you can order every core to do some work. So, the processor is an office where workers go to do their jobs. Therefore, in master-worker approach there is one master process (load coordinator) who has all the information about the underlying graph and sends data and tasks to the workers. As a result, we change from a program that computes everything serial this means that all jobs are queued and done sequential; to a program that is more con-

trolled because all the jobs are assigned to workers. In addition, this will be executed in the supercomputer sited on the University of Ljubljana, Faculty of Mechanical Engineering.

Nonetheless, we will see that this will cause some idle time due to communication issues.

For that reason, we tried to code one-side communication. We thought that if we delete the communication between nodes by creating an area that can be accessed by any of the workers without needing a master, we could increase the efficiency of the program. We will see that this is not completely true; but as with everything, if it is not tested, it is not known if it will work. This area that can be acceded for everyone is called "window".

Used Methods

In the introductory part we have explained the project motivation, which is the problem to be handled and the way we will manage to get improvement of the code. From now on, we will explain both approaches in detail, their advantages and disadvantages and the results we have obtained.

Master-worker approach: The best way to understand something is to see something that you understand and compare both. So, let's make an analogy: imagine a very big company with a lot of people working there, roles are assigned to rank the responsibilities of each worker. As a result, simple scale, we have a director or manager and workers or employees. The manager assigns various tasks to employees, which fulfil them and/or send new ones to coworkers and the process continues in this way. Then these jobs are sent back to the manager to get them together.

Our implementation is based on this logic. One process is selected as master, which has all graph instances, and sends job requests to other worker processes. When the worker processes finish their work, they send the results to the master process and wait for the new task. Master process is responsible for controlling the entire process and keeps track of the status of each worker. The master creates a vector of the solution, which is 0 or 1.

To communicate between load coordinator and workers *MPI Send* and *Receive* functions are used. In Figure 2 is a diagram that reflects this. Master worker is at the middle and the workers are connected to master to send and receive information.

In our particular case, the nodes sent by the master to the workers are the ones to branch and evaluate, and the nodes sent back to the master are the evaluated children of this node. To do this we have used c structures for nodes in which we allocate: a fixing nodes (which are already on the solution vector); fractional solution which is used to get the next branch; the level of the tree where is being evaluated; and the upper bound.

The master determines which node will be sent to which process and when the results will be received. Therefore, idle time occurs while processes are waiting for to send back the work done.

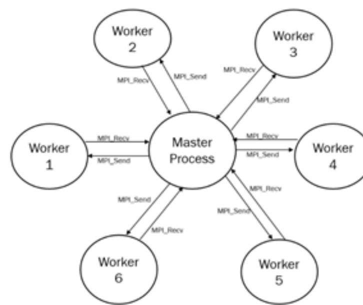


Figure 2: Diagram showing master-worker approach.

One-sided approach: Now that you are familiar to the company and you know well the roles that are assigned and the master-worker approach is well understood, let's restructure the company.

First of all, now there is no manager or director nor employees, everyone is responsible for its own work. Everyone is a manager and an employee at the same time. So, the idle time on processes waiting to send back the work done is no more a problem. You may think that this is like freelancers, and you are right; it is like a lot of people not related with the same porpoise: get the solution of max-cut probelm.

When one of the processes needs help by sharing one of its branch and bound nodes, other workers realise this and can get to its work directly, without waiting for the target process to send the message.

Accessing someone's work is done through the *MPI Window* allocated at the start of the algorithm. Note that this window is the determined area that other processes can reach and access data without involvement of the its owner. This section need to be implemented very careful to avoid race conditions. In our problem, this method is applied as shown in Figure 3.

All starts when process 0 branches the first node, and when it notices that it has more than one node in its queue, a free process reaches this node and evaluates it.

So, in this model, each processes can share its nodes with others. To get more points of view, we have developed two different versions for this access part shown in Figure 4.

As a consequence, we have version 1 which is the processes which are free inform to the others that are free, and allocates free nodes in his queue. But, version 2 processes finds which one is free and put the node in free process queue.

In order to send and retrieve data from the windows we used MPI Put and Get functions. However, if two processes access the same window at the same time it generates a race condition. This means that if they update the same value at the same time one of these values will be lost on the process.

Therefore, these operations needed to be executed in a way to prevent this situation. Managing this timing is what caused the idle time of the workers.

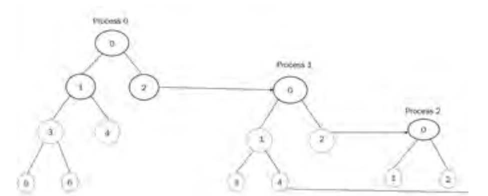


Figure 3: Shows the work of the one-sided approach.

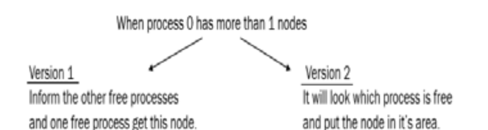


Figure 4: Two approaches for sharing information in one-sided communication.

Results

What we have done so far is to explain by facts the methods we have used and their behaviour. Nevertheless, these words mean nothing without some benchmarks to support them. Let's take a look at the numerical results of these approaches.

First of all, the benchmarks have been done by testing proposed parallel algorithms on 130 graph instances with a different number of vertices and edge weights. We took 5 instances for which the sequential branch and bound algorithm needed the longest time to obtain the optimum solution.

Notice that in Figures 5 and 6 ara tables displayed. These tables are the timing results of executing the master-worker approach and one-sided communication approach, respectively. All the graph instances contain 100 vertices.

These tables show the scalability results with both approaches when running the same instance with an increased number of workers.

As we can notice, from Figure 5 when increasing the number of processes the time decreases; being the serial version the one that takes longer times to solve the same problem.

Out of the number of running processes, one of them acts as a master process and the remaining workers evaluate the nodes. In other words, 1 master and 5 worker processes work in the case when 6 processes are assigned.

Process Numbers

3 prc	6 prc	12 prc	24 prc	48 prc	Serial
499.51	201.77	100.86	54.87	37.51	955.37
466.07	200.11	98.96	58.18	44.03	868.72
307.62	126.18	65.18	36.85	26.19	505.25
241.50	109.50	57.40	34.55	24.08	497.36
205.39	93.54	48.18	30.48	23.10	381.16

Figure 5: Reported timings of master-worker approach vs serial version.

Looking at the results of one-sided communication approach (Figure 6), occurs the same that in master-worker approach: we are getting a good improvement of the time that takes to solve the problem from serial to parallel version. Yet, they are not better than the results obtained on the master-worker approach.

3 prc	6 prc	12 prc	24 prc	Serial
603.09	420.36	244.83	144.58	955.37
555.70	416.58	283.15	165.99	868.72
348.91	271.99	141.50	95.71	505.25
310.18	210.45	145.50	87.25	497.36
256.95	171.44	123.53	72.60	381.16

Figure 6: One-sided vs Serial version

What's more, as a curious thing, we notice that if we use 48 processes or more the execution was like serial version. So, in that case we will not get any advantage of parallel communication when the workers increase. By way of comment, we think that this situation could be interesting to be study case.

Discussion & Conclusion

As we have been exposing, we proposed two different parallelization schemes of serial branch and bound algorithm, to solve a combinatorial problem. The first one is based on master-worker approach while the other utilises one-sided communication.

Looking at the results we can affirm that the parallelization was found to be successful, since the proposed algorithm could vastly reduce the computational time of the serial solver.

As we see, master-worker approach is more efficient when the problem is bigger, but when is not that big one-sided communication approach is also very useful.

It has been observed that, for the max-cut instances for which the optimum solution was obtained in short time, (i.e. the branch and bound tree is smaller) parallelization gives worse results as it requires more processing and more time is spent on communication. This means that if it is a small-sized problem is also suitable using the serial version; because the parallel version will not get much more advantage than serial.

Nonetheless, when we look at the five examples given in Figures 4 and 5, we see that the times are shortened in the two approaches compared to the serial version.

Although there is a constant decrease by looking at the tendency patterns, the results of the master-worker approach are better than the one-sided approach. The red line shows the result for the master-worker, while the blue line is the results of the one-sided approach. Results of the longest running instances were taken into account.

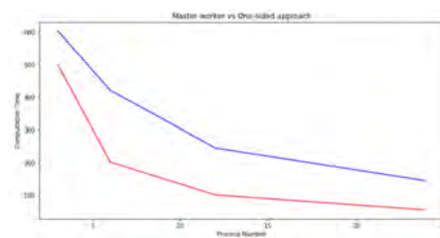


Figure 7: Graphical comparation between master worker approach and one-sided approach.

Despite the fact we thought that the one-sided approach would take less time than two-sided, the results did not come out as we expected. We think that the reason for this is that the processes need to do more control in order to prevent race conditions.

Acknowledgement

First of all,we'd like to thank our mentor Timotej Hrga for all support and interest through the two months, University of Ljubljana, Faculty of Mechanical Engineering for the use of their facilities. PRACE for giving us a place on the Summer of HPC programme.Many thanks to Leon Kos, Pavel Tomsic for all their work, in the first time online program they did everything well by organizing the training week, emails and weekly meetings .

References

[1] Franz Rendl, Giovanni Rinaldi, Angelika Wiegele. (2010). Solving Max-Cut to optimality by intersecting semidefinite and polyhedral relaxations

[2] Loc Q Nguyen. (2014). MPI One-Sided Communication

[3] William Gropp, Ewing Lusk, Anthony Skjellum.() Using MPI Portable Parallel Programming with the Message-Passing Interface

PRACE SoHPC Project Title

Implementation of Parallel Branch and Bound algorithm for combinatorial optimization

PRACE SoHPC Site

HPC cluster at University of Ljubljana, Slovenia

PRACE SoHPC Authors

İrem Naz Çoçan , Carlos Alejandro Munar Raimundo

PRACE SoHPC Mentor

Timotej Hrga, University of Ljubljana, Faculty of Mechanical Engineering

PRACE SoHPC Contact

İrem Naz Çoçan,Dokuz Eylül University
E-mail: iremnazcocan@gmail.com
Carlos Alejandro Munar Raimundo, University of Almería
E-mail: caamura@gmail.com

PRACE SoHPC Project ID

2020

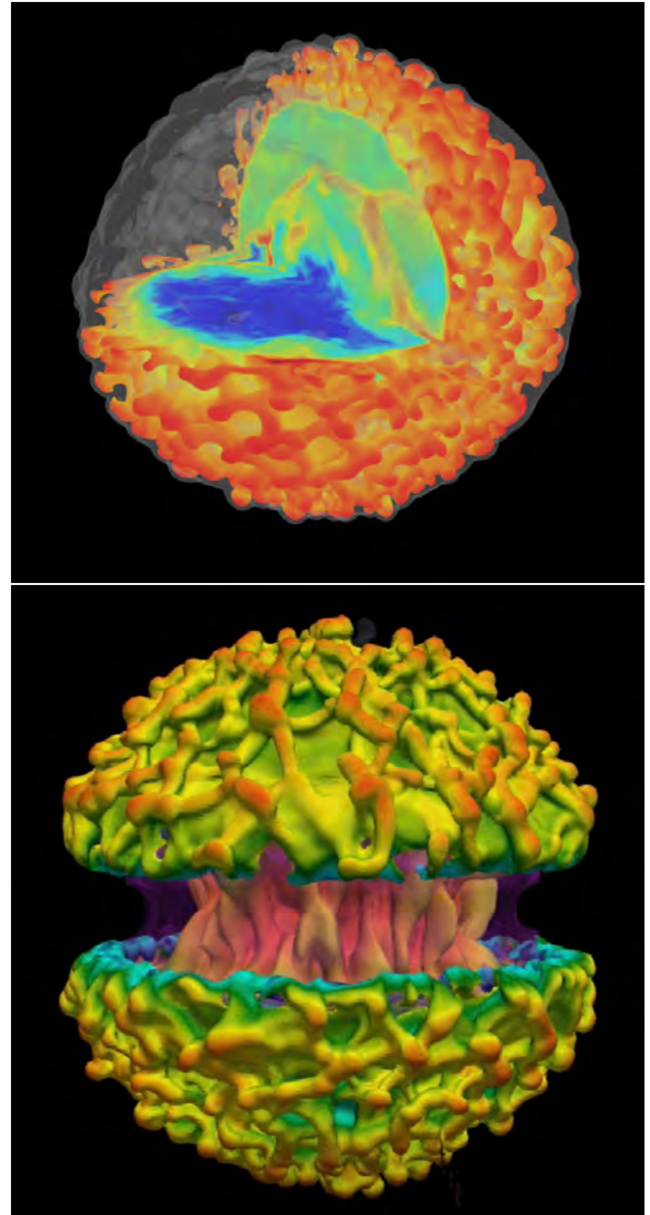
İrem Naz Çoçan

Carlos Munar

Visualisation of supernova explosions in an inhomogeneous ambient environment using the CINECA HPC facility in Bologna

Supernova Explosions Using HPC

Cathal Maguire & Seán McEntee



Recent developments in HPC has made it possible for the first time to simulate the full evolution of a massive star towards a highly-energetic supernova explosion, and also the subsequent expanding supernova remnant. Modelling these events can provide insight into the physics of supernova engines and can also explain the non-uniform distribution of heavy elements in the Universe.

Supernova (SN) explosions represent the violent death of massive stars and are one of the most energetic phenomena in the universe. These events are also the primary source of heavy elements in the universe.[1] When the star explodes, its matter is expelled into the surrounding environment, thus enriching it.

Supernova remnants (SNR's) are the outcome of SN explosions, and are diffuse extended sources with a rather complex morphology and a highly non-uniform distribution of ejecta. On SN explosion, the stellar material is ejected from the star, and travels freely until it reaches the circumstellar medium (CSM). The CSM contains the mass-loss from the progenitor star in the years leading up to the explosion.

When the expanding SN ejecta interacts with the CSM, some of the material will continue to propagate outward in what is known as the forward shock. Conversely, some of the material will travel backwards into the freely expanding ejecta after colliding with the CSM. This is known as the reverse shock.

The remnant morphology reflects, on one hand, the highly non-uniform distribution of ejecta due to pristine structures developed soon after the SN, as well as the imprint of the early interaction of the SN blast with the inhomogeneous circumstellar medium (CSM).

In Figure 1, we can see a schematic diagram showing the density profile of a spherically symmetric blast wave expanding in a uniform ambient environment. This represents a 2D slice of our 3D model, and the different regions are easily distinguishable, such as the reverse and forward shocks that were mentioned earlier.

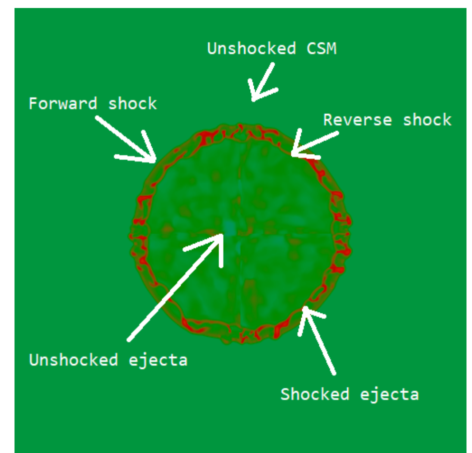


Figure 1: Schematic diagram of a spherically symmetric SNR in a uniform ambient environment at a time of 1,000 years after SN explosion.

What is also noticeable is that within

the dense mixing region (shown in red), we can see the fingerprint of hydrodynamic instabilities developing. SNR's can span for 10's of parsec (1 parsec = 3.3 light-years) and are observable for thousands of years,[2] with this particular snapshot taken 1,000 years after SN explosion.

Motivation

Our project's aim was to develop complex SNR models with key features that are present in real-life, observable remnants. This was achieved by introducing asymmetries into our models, and thus expanding on the spherically symmetric case depicted in Figure 1. Asymmetries in SNR's offer the possibility to probe the physics of SN engines by providing insight into the anisotropies that occur during the SN explosion. These asymmetries also offer the possibility to investigate the final stages of stellar evolution by unveiling the structure of the medium immediately surrounding the progenitor star.

One SNR that we investigated was the case of Cassiopeia A, which exploded approximately 350 years ago[3] at a distance of 3.4 kpc away from Earth.[4] In this case, asymmetries developed within the blast wave shortly after explosion while the ambient environment was approximately uniform. This formed the basis of our first asymmetric simulation, which we denoted as 'Model A'. The details about the implementation of the asymmetries in our models will be discussed in the next section.

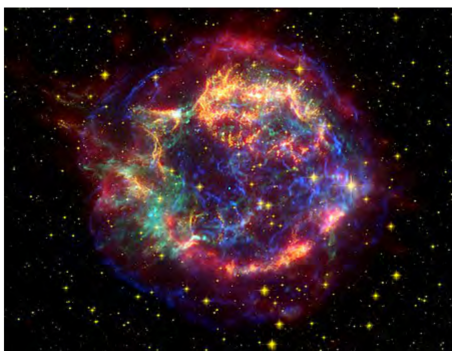


Figure 2: Cassiopeia A (Courtesy NASA/JPL-Caltech).

Another SNR that was of interest to us was the case of SN1987A. This supernova was observed on February 23rd, 1987, by two independent astronomers, Ian Shelton and Albert Jones.[5] It was also the brightest and nearest Supernova to occur in roughly 350 years, and thus has since been the subject of intense investigation and modelling. In this remnant, the initial blast wave was approximately spherically symmetric, while the asymmetries were present in the surrounding CSM.[6] Figure 3 shows a ring structure surrounding SN1987A, and we attempted to emulate this in our second asymmetric model which was denoted 'Model B'.

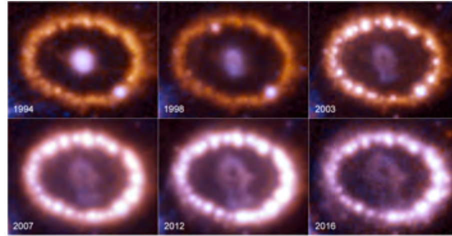


Figure 3: SN1987A (Credit: NASA/ESA, R. Kishner, and P.Challis).

Methods

All the simulations conducted in this project were run on the GALILEO system at the CINECA facility in Bologna (shown below). This supercomputer contains 1022 nodes, with 36 cores-per-node. Nodes are individual computers that consist of one or more CPUs (Central Processing Units) together with memory. For comparison, a Personal Computer is considered to be a single node and has a CPU with 4 cores.

Figure 4; Galileo supercomputer rack at CINECA Source: https://en.wikipedia.org/wiki/Galileo_(supercomputer)

The supernova explosion models were run using The PLUTO Magnetohydrodynamic Code,[7] a freely-distributed software for simulating astrophysical fluid dynamics (see Software Applied & More Information). The input parameters for the code were: the mass of the progenitor star, the initial remnant radius, the forward shock velocity, the density of the CSM, and also the pressure of both CSM and ejecta. The outputs of PLUTO were the velocity, pressure, and density distributions at different stages of the evolution over a duration of 2,000 years.

Model A was created by introducing dense, high velocity clumps within the initial remnant to mimic post-explosion anisotropies of Cassiopeia A. Model B was constructed by introducing two inhomogeneous features of high-density in the ambient environment: the first was a torus, or 'doughnut-shaped' feature encircling the blast wave, such as that present in SN1987A. A molecular cloud of high-density was also introduced, and this feature was offset against the origin of the blast wave.

Once the 3D simulations were completed, data visualisation of the 2D slices of our models was performed using the Interactive Data Language (IDL).[8] To get a better understanding of the models, we employed the 3D visualisation software Paraview. Our final models were also uploaded onto SketchFab, a platform for publicly sharing 3D models.

Results

Once we were happy with the efficiency of our code, as well as the final structure of our models, we began comparing the symmetric and asymmetric cases. In order to probe the inner and outer structure of the models, two-dimensional slices of the three-dimensional density profiles were taken, at various stages in the models' evolution. In doing so, we could examine the models' diverse regions of high and low density, caused by the abnormalities introduced in the asymmetric cases.

The comparison between the spherically symmetric case and Model A is shown below in Figure 5. Notice the voids of low-density in Model A on the left. These are produced as a result of the propagation of the initial clumps towards the mixing region. These clumps also appear to cause a slight protrusion of the outer remnant, distorting the left side of the remnant in the process.

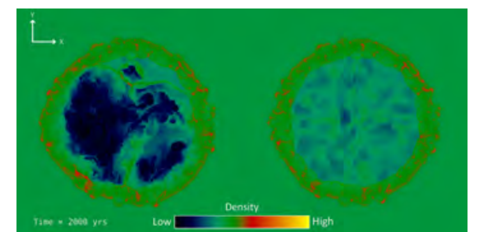


Figure 5: A 2D slice at the centre of Model A's 3D density profile (left), alongside a 2D slice at the centre of the symmetric model's 3D density profile (right), both at t = 2000 years.

This figure shows the density structure of the centre of the model approxi-

mately 2,000 years after the initial stellar explosion. These regions of low density, or voids, relative to the symmetric case, are a key feature which is known to exist in Cassiopeia A.

The effects of the asymmetries introduced in the surrounding environment of the expanding blast wave, such as that present in Model B, were also evident in the final model's density structure. Clear regions of high density arose from the interaction of the blasted ejecta with the regions of high density (i.e the torus and cloud structures) in the blast wave's ambient surroundings. These regions of high density, relative to the unshocked ejecta, are shown in Figure 6. We can clearly see strong interaction between the blast wave and the high density regions, resulting in clusters of relatively high density in these areas.

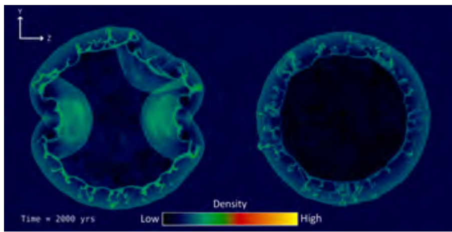


Figure 6: A 2D slice at the centre of Model B's 3D density profile (left), alongside a 2D slice at the centre of the symmetric model's 3D density profile (right), both at t = 2000 years.

These regions of high density are analogous to the torus-like feature encircling SN 1987A. This occurs when the expanding blasted ejecta interacts with material which was previously ejected into the CSM from the progenitor in its final stages of evolution.

The final structure and morphology of Model A and Model B may be difficult to visualise from the figures above, therefore we have also uploaded our final models for public access on SketchFab.com. The links to the respective models can be found in the Appendix, but examples of what you can expect to see are given in Figures 7 & 8 below.

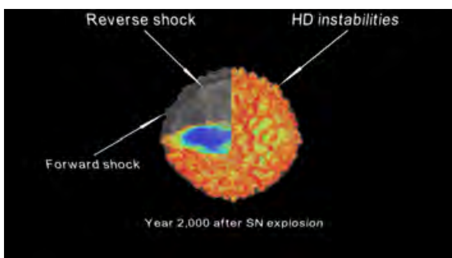


Figure 7: 3D visualisation of the density profile of Model A at a time of 2,000 years after SN explosion.

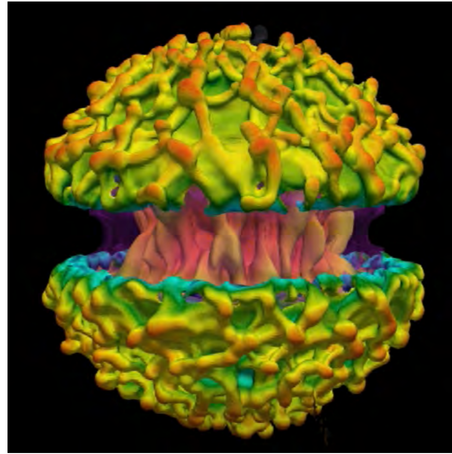


Figure 8: 3D visualisation of the density profile of Model B at a time of 2,000 years after SN explosion.

Discussion & Conclusion

The significance of asymmetries, either within the blast wave, or in it's surrounding environment, can clearly not be understated. As outlined in the results above, the morphological and structural differences of the two models with respect to the symmetric cases is evident.

Figure 5 shows clear regions of low density ejecta emerging within the confines of the blast wave of Model A, whereas Figure 6 shows clusters of high density resulting from the interaction of the blasted ejecta with regions of high density in the surrounding CSM. These two models clearly display the impact of structural asymmetries, at early stages of a SN's evolution, on the final morphology of the resultant SNR.

Knowing the result of these effects, one could input real-world, observational parameters of Supernovae such as Cassiopeia A or SN 1987A, into the framework of our models, and attempt to evolve these models forward in time, in order to determine how they might look hundreds or thousands of years into the future. An evolutionary model such as this would also give us an insight into the various conditions and processes which produce the majority of the heavy elements in the Universe, via the r-process in supernova nucleosynthesis.

Acknowledgements

We would like to extend our utmost gratitude to our supervisor, Prof. Salvatore Orlando, as well as all the very helpful staff at CINECA, in particular Massimiliano Guarrasi. We would also like to thank the coordinators of the Summer of HPC 2020, and PRACE, for providing us with not only a unique opportunity, but also an exceptional learning-curve.

References

[1] Burrows, A. (2000). Supernova explosions in the universe. *Nature, 403(6771), 727-733.*

[2] Reynolds, S. P. (2008). Supernova remnants at high energy. *Annu. Rev. Astron. Astrophys., 46, 89-126.*

[3] Stover, D. (2006). Life In A Bubble. *Popular Science, 269(6), 16.*

[4] Fesen, R. A., Hammell, M. C., Morse, J., Chevalier, R. A., Borkowski, K. J., Dopita, M. A., ... & van den Bergh, S. (2006). The expansion asymmetry and age of the Cassiopeia A supernova remnant. The Astrophysical Journal, 645(1), 283.

[5] Kunkel, W., Madore, B., Shelton, I., Duhalde, O., Bateson, F. M., Jones, A., ... & Menzies, J. (1987). Supernova 1987A in the large magellanic cloud. *IAUC, 4316, 1.*

[6] Dwek, E., Arendt, R. G., Bouchet, P., Burrows, D. N., Challis, P., Danziger, I. J., ... & Slavin, J. D. (2010). Five years of mid-infrared evolution of the remnant of SN 1987A: The encounter between the blast wave and the dusty equatorial ring. *The Astrophysical Journal, 722(1), 425.*

[7] Mignone, A., Bodo, G., Massaglia, S., Matsakos, T., Tesileanu, O., Zanni, C., & Ferrari, A. (2007). PLUTO: a numerical code for computational astrophysics. *The Astrophysical Journal Supplement Series, 170(1), 228.*

[8] Bowman, K. P. (2006). An Introduction to Programming with IDL: Interactive data language. *Elsevier.*

Appendix

Final Morphology of Model A:
`https://rb.gy/5p37jf`

Final Morphology of Model B:
`https://rb.gy/lknxwu`

PRACE SoHPC Project Title
Visualisation of supernova explosions in a magnetised inhomogeneous ambient environment

PRACE SoHPC Site
CINECA, Italy

PRACE SoHPC Authors
Cathal Maguire & Seán McEntee
Trinity College Dublin, Ireland

Seán McEntee

PRACE SoHPC Mentor
Prof. Salvatore Orlando, CINECA, Italy

PRACE SoHPC Contact
Name, Surname, Institution
Phone: +12 324 4445 5556
E-mail: leon.kos@lecad.fs.uni-lj.si

Cathal Maguire

PRACE SoHPC Software applied
PLUTO, Paraview, Blender, and SketchFab

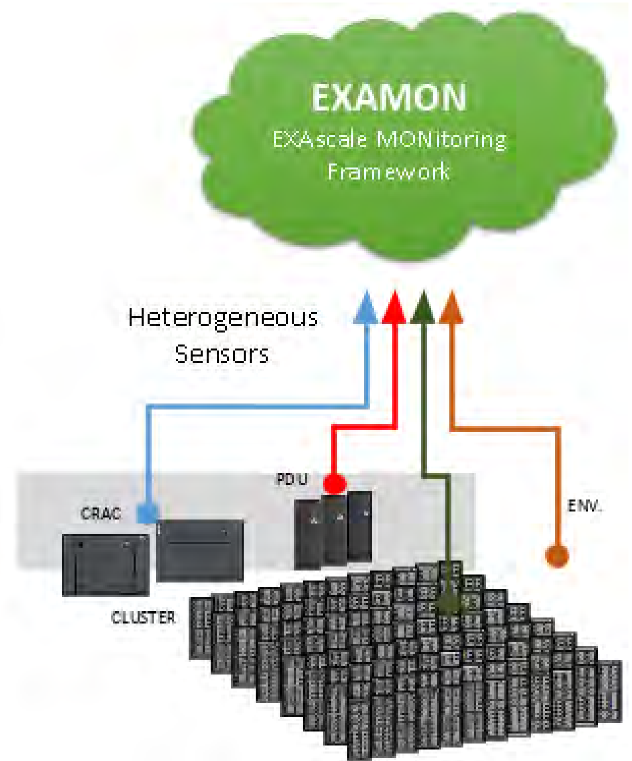
PRACE SoHPC More Information
PLUTO User Guide
Paraview
Blender
SketchFab

PRACE SoHPC Project ID
2003

Anomaly Detection in High Performance Computing Systems

Nathan Byford
Stefan Popov
Aisling Paterson

Energy efficiency and power usage are the main concerns for the effective operation of a HPC cluster infrastructure. This work describes the creation of visual dashboards through the usage of Bokeh, Grafana and Redash to monitor the state of HPC clusters and the retrieval of a data set with Apache Spark's package Delta Lake.



1 Background

The development of exascale high performance computing (HPC) systems brings enormous potential to accelerate progress in scientific research. Protein docking, molecular modelling and simulation of physical phenomena are just a few computationally-expensive applications which are accelerable through the use of HPC. However, with the development of such powerful machinery come ineluctable hurdles. In the context of supercomputing, this means limitation of peak performance by extensive power consumption, and challenges in system management due to fallible software and hardware.

While performance limitation due to power consumption necessitates large changes in infrastructure, the detection and analysis of system failures presents a much more manageable task. The inspection of particular properties of the system, such as power consumption, CPU and GPU temperature, and drain status can all help one identify system failures and better our understanding of the system's behaviour. This understanding is invaluable to the design of more reliable, energy efficient computing systems in the near future.

At the CINECA database, the development of an exascale monitoring infrastructure, named ExaMon, entails collection of data of two types. Firstly, the collection of physical parameters of system components, such as the temperature of a particular CPU, and further, workload information, for example the computing resource requirement for a particular user. ExaMon is able to collect up to 70 GB per day of telemetry data from the CINECA supercomputer system.[1] The vast amount of data collected present great potential for machine learning monitoring techniques and automation of the data centre management process.

One example where these techniques are especially applicable concerns data on the power consumption of previous jobs run on the supercomputer system. With large data sets ideal for training, machine learning models, based on a random forest, allow for prediction of the power consumption of a HPC job with good accuracy, proving 9% average error of predicted against experimentally observed power measurements.[2]

Another aspect of data centre monitoring process is custom modification of the software and hardware used. That approach can discover power variations at the cluster and node level down to the millisecond scale [2]. And, it is achieved with no impact on the computing resource availability as the power monitoring is carried out outside the computing nodes of the cluster.

Another use of machine learning techniques lies in the detection of anomalies in system operation. Identification of faults in a supercomputer system composed of many various components is a demanding task, however, a model which, following training, can recognise healthy versus faulty states in components presents the opportunity for automated anomaly detection. Previous data collected by Examon was used for a semi-supervised model which created a model of the normal state of the supercomputer system, allowing for determination of anomalous system states.[2] Such anomalous

states and the metrics which allow us to monitor them are outlined in this research.

2 Methods

In order to infer suitable parameters to track for anomaly detection, the first task was to visualise data from Marconi 100, the new cluster that is part of CINECA's exascale supercomputer. The ExaMon framework is depicted in Figure 1.

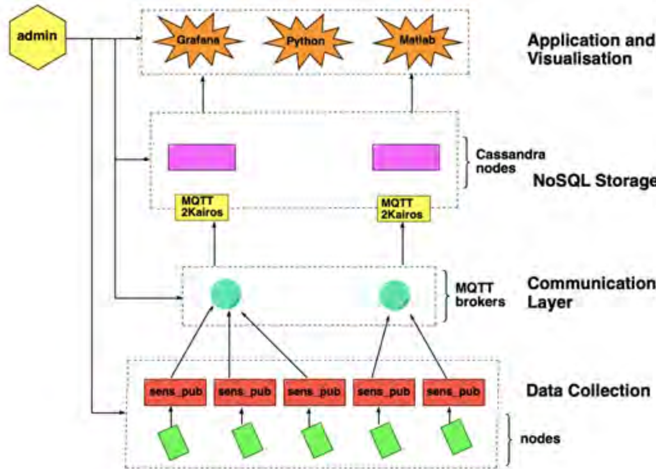


Figure 1: Exascale Monitoring Framework

Investigation into the data collected on Marconi 100 utilises the tailored query language ExaMon query language (ExamonQL), to obtain data from MQTT brokers. MQTT brokers are bound by the 'publish-subscribe' messaging pattern, whereby as information becomes available from sens_pub, the plugin which will publish the data, it is received by the broker and passed onto a subscriber, here MQTT2Kairos. Following the communication layer, the data can be stored by the time series database KairosDB. This database is built on the NoSQL database Apache Cassandra nodes. Finally, the collected data is available for manipulation and visualisation. An ongoing theme in this project concerns the visualisation of data collected by the ExaMon framework in Grafana, a web application for the design of monitoring dashboards illustrated with such data, with capability for an interactive user interface and updates in real-time.

Development of the monitoring dashboard necessitated prior generation of plots in a web page. Bokeh was used to provide this server and deliver the output of the ExamonQL queries which demanded information on a given metric. The plots served by Bokeh were made possible *via* a Python script, containing ExamonQL queries, which retrieves the data through a KairosDB server. The Bokeh server with its specific HTML was then used to load content into the Grafana dashboard.

Similarly, another aspect of the project was to integrate the data visualisation tool Redash into the ExaMon framework. This would lie amongst the other current application and visualisation tools as shown in Figure 1. This allows visualisation of the time series data of the Marconi 100, with particular focus on job-scheduling data. Predictions made by an anomaly detection machine learning could be usefully visualised also through Redash.

A particularly useful feature of Redash is its flexibility for queries in the query language native to the data source. As mentioned, the data on the Marconi 100 utilises the tailored query language ExaMonQL. To integrate Redash with the ExaMon framework it was necessary for Redash to be able to gain access to and read from the NoSQL data storage used by the ExaMon framework, namely the KairosDB database. This involved the creation of a *query runner* from Redash to the data source, which instructed Redash how to access KairosDB as well as connecting the query language ExaMonQL.

A third aspect of the project was to investigate the possibility of training a model for detecting anomalies in real-time and alerting the system administrators. To that end, we first had to obtain a data set from the ExaMon framework. We have chosen to transfer the ExaMon data for this task into a more suitable format. We have used Delta Lake, a package within Apache Spark that brings ACID transactions to its processing engine. Delta Lake offers numerous features and advantages to Spark. Most importantly, it guarantees schema enforcement and evolution, meaning that it prevents data corruption through insertion of illegal values and offers to possibility for the data to "evolve", i.e. with time, there may be some other columns we might want to add to our table and Delta Lake enables this seamlessly. Another great aspect of Delta Lake is that it utilises the Apache Parquet format for storing the data. Apache Parquet is a column-oriented format, introducing much faster seek time for analytical queries and efficient and effective compression schemes, effectively reducing the amount of storage needed compared to the traditional row-based format in ExaMon. Moreover, Delta Lake does not make distinction between table and stream read or write queries: its tables can be used for both scenarios with no overhead. Figure 2 illustrates the architecture of Delta Lake.

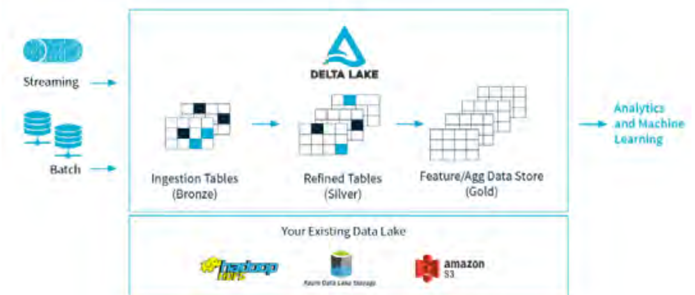


Figure 2: Delta Lake architecture. Taken from delta.io

Here, we can observe the different logical views of tables, Delta Lake's compatibility with major cloud storage solutions (Microsoft Azure, Amazon S3, and Hadoop) and its ignorance towards the batch and streaming setting. As such, Delta Lake is one of the most popular packages within Apache Spark and is being used by many companies worldwide.

Consider for example a question that might arise from this data: Plot the temperature of node X for May through August 2020 and check if there are any anomalies (unusually high or low values) in it? In the traditional row-oriented format, the processing engine would have to read a batch of rows, and for each row it should scan up to it's N-th column (where the temperature is stored) and this could take a while, considering that there might have been many columns in between that can be large in memory and force the engine to read more disk blocks.

3 Results

The graphic in Figure 3 is exemplary of a time series plot for a given metric over a specified time period, here the temperature of a CPU of a particular node of the Galileo cluster of Marconi 100 over a period of two days.

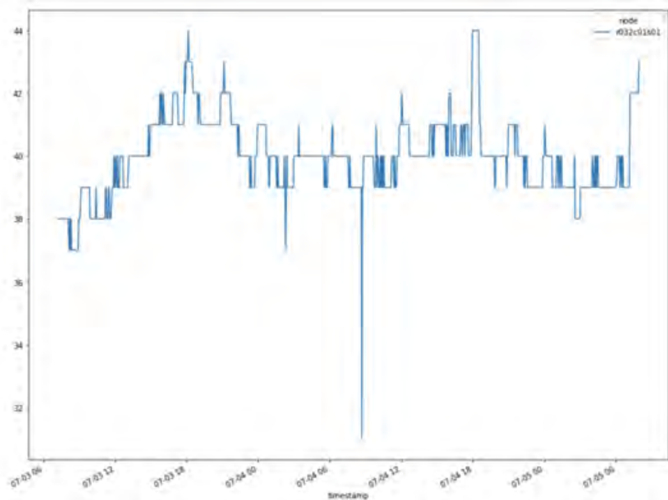


Figure 3: Time Series for the Metric CPU1_Temp

The time series plot shows fluctuations in CPU temperature, with a large drop in temperature during the morning of 04/07/2020. This could indeed be the kind of anomaly characteristic of a system failure which the machine learning algorithms should observe.

Similarly, the graph in Figure 4 below displays of the frequency of job failures for the nodes of the Galileo cluster of Marconi 100 over a period of a day.

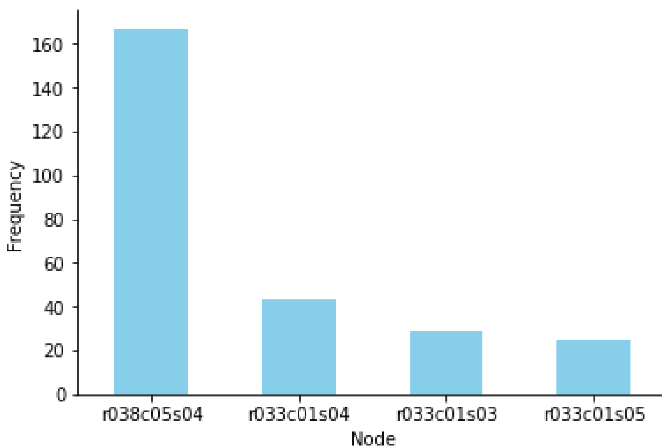


Figure 4: Frequency of job failures for given nodes

Approximately 168 metrics are being monitored for the Marconi 100 cluster in the summer of 2020. Many of these metrics are associated with the environmental parameters of the hardware, but there are others that concern the jobs that users submit to it. The task of exporting Marconi 100 data to Delta Lake is rather straightforward (or is it?). We just query ExaMon and save the data returned from it into Delta Lake table. Sounds great, (almost) doesn't work. As already mentioned, Delta Lake works on top of Spark, which is optimised for distributed data processing. But being remotely connected to a HPC machine and from there remotely accessing a data source and saving it to a third remote place is obviously a recipe for problems. And problems there were. From having reached disk quota limit, getting schema specification errors, waiting for SLURM resource allocation, reaching wall-time on the submitted job and restarting from a checkpoint, to finally carrying out the final execution, I had ran into quite some problems during this task.

The final result is a 106 GB of Marconi 100 data from 1st of May to 3rd of August 2020 in two tables. One of the tables concerns numerical data, such as temperature values, memory allocation, number of idle/allocated/drained/downed nodes/CPUs etc. and another for string logs from the different plugins that are set up within ExaMon. Because Delta Lake saves data in Parquet format (column-oriented), each column value is stored right next to each other and reading the column data can be very fast (fewer disk reads) for the analytical queries that we will be are dealing with. Also, took up less storage space for the data, because there are techniques for compressing it (each column contains homogeneous data). In big data scenarios, column-oriented storage is the way to go.

References

[1] A. Bartolini, A. Borghesi, F. Beneventi, A. Libri, D. Cesarini, L. Benini and C. Cavazzoni, Paving the Way Toward Energy-Aware and Automated Datacentre, ICPP 2019, August 5–8, 2019, Kyoto, Japan, doi:10.1145/3339186.3339215

[2] A. Bartolini, A. Borghesi, A. Libri, D. Gregori, S. Tinti, C. Gianfred and P. Altoè, The D.A.V.I.D.E. Big-Data-Powered Fine-Grain Power and Performance Monitoring Support. In Proceedings of the 15th ACM International Conference on Computing Frontiers, CF 2018, Ischia, Italy, May 08-10, 2018 (2018), pp. 303–308.

PRACE SoHPCProject Title

Anomaly Detection of System Failures in HPC-Accelerated Machines Using Machine Learning Techniques

PRACE SoHPCSite

CINECA, Bologna, Italy

PRACE SoHPCAuthors

Nathan Byford, Imperial College London, UK.
Stefan Popov, International Postgraduate School Jožef Stefan, Slovenia
Aisling Paterson, Trinity College Dublin, Ireland

PRACE SoHPCMentor

Andrea Bartolini, University of Bologna, Italy
Andrea Borghesi, University of Bologna, Italy
Francesco Beneventi, University of Bologna, Italy

PRACE SoHPCContact

Stefan Popov, International Postgraduate School Jožef Stefan, Slovenia
E-mail:popovstefan@live.com

PRACE SoHPCSoftware applied

Virtuoso

PRACE SoHPCMore Information

www.virtouso.org

PRACE SoHPCAcknowledgement

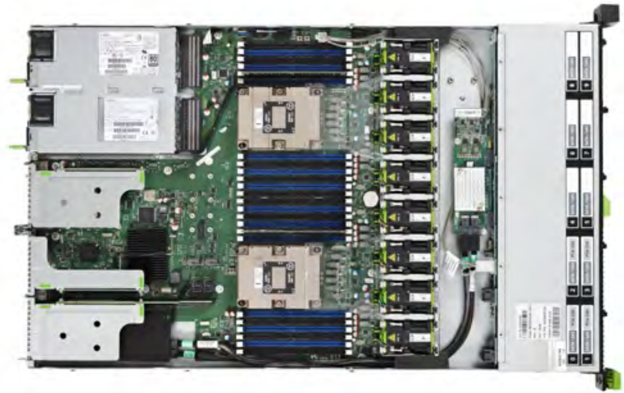
We are thankful to all the mentors and colleagues from the University of Bologna for their help, guidance and support throughout this project.

PRACE SoHPCProject ID

2004

Exploit persistent memory to improve existing Charm++ fault tolerance

Persistent Memory checkpoint



Petar Dekanović, Roberto Rocco

Charm++ is a message passing framework with multiple features, including fault-tolerance. Instances can be saved to disk, but if it is replaced by persistent memory, it might be possible to maintain the same functionality with major gains in terms of the performance.

Fault tolerance is the field of distributed computing aimed at the design of algorithms and systems able to deal with faults. In the past it used to be not related to High-Performance Computing (HPC) but, due to the evolution of the architectures, nowadays more and more efforts are exploring the intersection of the two fields.

Charm++ checkpointing scheme

It used to be irrelevant before since failures happened rarely in a controlled environment typical of HPC. But, with the increase in the number of machines which led to the enlargement of this eventuality, it must now be considered: programs are likely to encounter one fault (at least) and must be able to handle them.

This fact has been analyzed by other efforts in the field: Schroeder and Gibson[1] have collected data at two large high-performance computing sites, showing failure rates from 20 to more than 1000 failures per year. Future systems will be hit by error/faults much more frequently due to their scale and complexity.[2]

Fault tolerance can be achieved following many different approaches. Among all the efforts produced, the most used technique used for implementing fault-tolerance in an application is Checkpoint and Restart (C/R). It consists of the periodic creation of saving points from which the execution can safely restart in case of failure. While standard C/R doesn't need to deal with the continuation of the application (upon fault, execution will stop), online C/R goes further: it allows the program to recover from the failure right away and proceed like nothing bad happened. Nowadays, more and more high-performance programs are using some kind of system for C/R, and Charm++ is one of them.

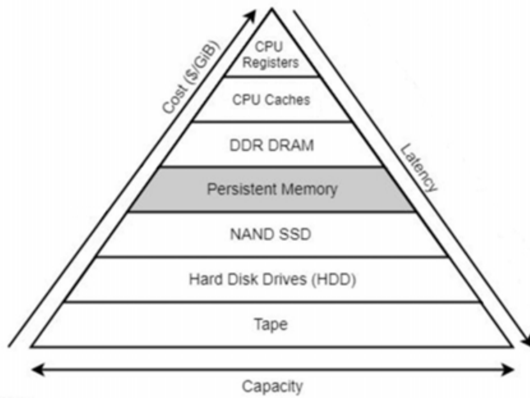
Charm++ is a parallel programming framework written in C++ based on abstracting fundamental units of sequential code which can be executed simultaneously into Chare objects. Each chare encapsulates its state and communicates with other chars by invoking their so-called entry methods: this can be seen as asynchronous message exchanges. The framework has been in production use for over 15 years and has been used by multiple successful applications. It contains multiple features ranging from automatic load balancing to online C/R which aren't common in other similar frameworks.

The main drawback of using C/R is the overhead it introduces: the status must be written to persistent storage periodically which heavily impacts the performance of the application. The focus of this effort is to try to leverage the recent improvements in the persistent storage field to reduce the mentioned overhead.

Persistent Memory

Data storage plays a core role in every computing system: without it, it would be impossible to do any meaningful operation. The field evolved by specializing in different tasks, depending on the characteristics it must provide to

the system. This, as a consequence, created a fracture: on one side there is fast storage able to keep up with the speed of the processor; on the other, there is reliable one able to maintain its information even after power-off. The first led to the development of various kind



Memory hierarchy diagram

of volatile memory: processor register, caches and main memory (also known as RAM – Random Access Memory). The other produced devices like magnetic tapes, HDDs (Hard Disk Drives), SSDs (Solid State Drives).

This basic scheme hasn't been changed since the beginning of the era of modern computers and although devices from the first group had become more spacious and the second group had become faster with the introduction of SSD technology, there always was a huge gap between these two. Persistent Memory (PMEM) tries to tackle this everlasting problem by promising performance near main memory and persistence like disk devices.

Usage

Persistent memory, due to its heterogeneous nature, exposes the possibility to use the storage in different ways depending on the use-case. Intel's representative, Optane Persistent Memory, offers two distinct run configurations:

- Memory Mode – uses persistent memory as volatile main memory, where existing RAM functions like slower cache;
- App Direct Mode – uses persistent memory as a permanent data store that is available like any other disk device.

The first configuration immediately gives a transparent increase in terms of main memory size that can reduce page swapping for programs with large memory footprints. However, it reduces the overall memory responsiveness due to slightly slower speed compared to RAM which should be considered when choosing this option.

The other configuration achieves data persistency while providing bandwidths multiple times larger than other existing disk options.

The second mode also allows to access the storage in different ways, like FSDAX (File System Direct Acc(X)ess), where the device will be accessed using standard file I/O operations; or PMDK (Persistent Memory Developer Kit) which fully exposes device hardware to a programmer and allows customizing applications for the most performance.

Changing Charm++

Persistent Memory can be configured and used in a lot of different ways depending on the specific need. The first mode, although interesting, was shown to be irrelevant simply because of the fact it lacks a crucial feature – persistence. On the other hand, App Direct Mode, which turns the device into a really fast non-volatile memory, proved to be the one useful. With that, both FSDAX and PMDK approaches for memory access became viable options and were exploited during the implementation.

Being able to understand and utilize this new kind of technology has been only half of the story. The other has probably been even more challenging due to a necessity for deeper exploration of the already rounded system of checkpointing to find a way to incorporate new approaches.

As with every other big system, a great chunk of time was spent on code analysis, but also on comprehending the ways the fault-tolerance module is entangled and connected with other parts of the framework. This turned out to be very important especially when the PMDK approach was introduced.

Performance results

The performance of the changes has been evaluated by running a built-in stencil application that uses the Jacobi method to determine the solutions of a strictly diagonally dominant system of linear equations. This application periodically stores a checkpoint, also printing the time needed for the operation.

The test has been performed many times with different configurations which consist of the number of chares participating in the computation and number of processors and nodes that are available to the application. Test features all of the newly added checkpointing schemes alongside the ones already present. Worth mentioning is there is also the `O_DIRECT` mode, which ignores default disk caching.

Graphs in Figure 1 show the results obtained with all the hardware configurations. The number of application chares and time needed to checkpoint (in logarithmic scale) are represented on the axis.

All modes tend to follow the same pattern with configuration changes and also show the same order in terms of performance: Memory is always the fastest, followed by the two from persistent memory, and at the end are the disk (with `O_DIRECT` being very far behind). The closeness of the methods leveraging persistent memory can be analyzed: usually FSDAX tends to be slower than PMDK due to relying on the operating system as a mediator for all operations. The overhead impact can be reduced by performing operations on big chunks of data, and checkpointing tends to do so.

Performance of the disk can also be analyzed: the `O_DIRECT` mode achieves really bad times compared to all the other solutions, while the normal disk mode tends to have similar performance with persistent memory. However, only `O_DIRECT` mode is completely persistent: normal disk uses memory as cache, and in case of failure all the data cached and not flushed will be lost, leaving obsolete information in

Persistent memory module

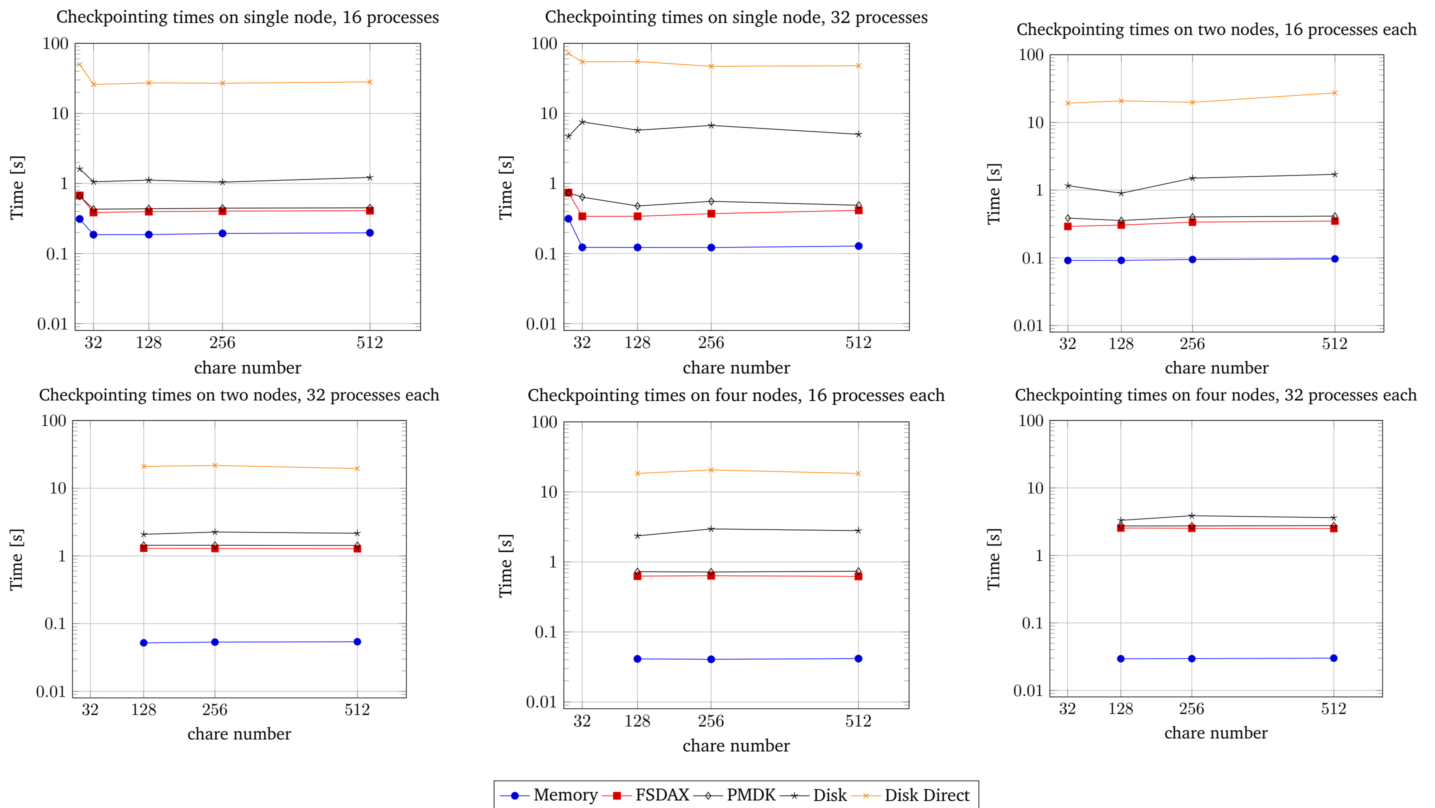


Figure 1: The graphs represent the results obtained with the tests under the specified configurations.

the persistent storage. This proves that the disk alone cannot reach both speeds comparable to memory and persistency, while persistent memory can do so.

Future work

All the performance obtained with checkpointing via persistent memory can be used to introduce new functionalities into the framework. Transactionality is one of those. It is based on ACID properties - a set of constraints that limit the effect of a failure in a system. The name comes from the acronym of Atomicity, Consistency, Isolation, and Durability: the first ensures that each operation is treated as single even when composed by sub-parts, the second assures that only consistent states are reached, the third states that operation are independent of each other and the last guarantees that all the changes caused by the operations won't be lost.

The concept of transactionality comes from the database field and hasn't been used often outside of it due to the strictness of the constraints. To introduce transactionality into a checkpointing system, process execution must be treated like a query in a database: all the messages exchanged must be considered alongside the shared data manipulation. In Charm++, there is no shared data, so the focus is only on message passing: given this point, the introduction of transactionality can be obtained by sending all the messages of a chare only when execution is completed successfully. Checkpointing shall be used to ensure that the messages are delivered correctly and without missing anything.

This effort managed also to produce the first version of transactional checkpointing, in which it was possible to postpone the sending of all the message to the end of a chare, ensuring that no message is sent in case of failure. This should be enough to introduce transactionality with the addition of asynchronous checkpointing, but this last feature proved to be hard to implement: Charm++ doesn't support it by default and a deeper analysis of the framework is needed.

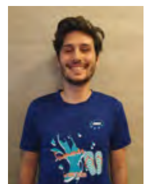
Transactional checkpointing, while being difficult to incorporate, introduces functionalities that may become useful for today's and future HPC developers. It may be possible to exploit the performance obtained with persistent memory to realize this feature with acceptable overhead in the future.

References

[1] B. Schroeder and G. A. Gibson, (2010). A Large-Scale Study of Failures in High-Performance Computing Systems

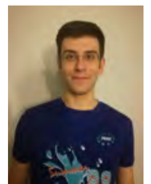
[2] Cappello, Franck (2009). Fault tolerance in petascale/exascale systems: Current knowledge, challenges and research opportunities

PRACE SoHPCProject Title
Charm++ Fault Tolerance with Persistent Memory


Petar Đekanović

PRACE SoHPCSite
EPCC,
University of Edinburgh,
The United Kingdom

PRACE SoHPCAuthors
Petar Đekanović,
University of Belgrade,
Serbia
Roberto Rocco,
Politecnico di Milano,
Italy


Roberto Rocco

PRACE SoHPCMentor
Dr. Oliver Thomson Brown,
EPCC,
The United Kingdom

PRACE SoHPCContact
Petar Đekanović, University of Belgrade
Phone: +381 65 2526 445
E-mail: petar.djekanovic@gmail.com
Roberto Rocco, Politecnico di Milano
Phone: +39 333 9955 469
E-mail: roberto2.rocco@mail.polimi.it

PRACE SoHPCMore Information
If you need any additional project-specific information, do not hesitate to contact any author (Petar or Roberto) as well as Oliver, the mentor.

Hardware was provided for this project by NEXTGenIO. The NEXTGenIO system was funded by the European Union's Horizon 2020 Research and Innovation programme under Grant Agreement no. 671951.

Other than that, you can find useful information on this and other interesting topics about high-performance computing at Summer Of HPC and PRACE sites.

PRACE SoHPCProject ID
2005

ARM tooling

Irem Kaya and Jerónimo Sánchez

Through the last years, ARM has proven to be a competent computer architecture for servers. This project aims to improve the ecosystem in which researchers are working, providing them with tools that right before this affair were not ported to ARM or, simply, did not exist.

In recent times, the ARM server application ecosystem has been explored in order to help ARM to reach its full potential on said space.

Notwithstanding that this analysis has proven that the ecosystem is rich, it is not mature enough for the common usage on servers, thus is not attractive for companies.

Owing to this vast issue, there has been an exponential growth in the number of research facilities about this obstacle around both the fields of computer architecture and applications porting.

As an outcome of this R&D, three convenient proposals have been developed throughout the last few months on the **EPCC**, resulting on the following projects: **Firestarter**, **IOR-parser** and **dmgler**.

Each of this utilities has run on the Fulhame cluster which consists of 64 compute nodes, each of them with 32 cores an up to SMT4, based on the Cavium ThunderX2 ARM processor; composing the cluster with up to 8192 hardware threads and 128 GiB of RAM[1].

As shown in the Figure 1, each computer of the Fulhame cluster is made of 2 compute nodes.

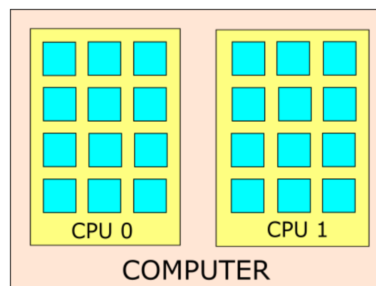


Figure 1: Fulhame computer blade scheme.

Firestarter

Firestarter is an open source tool that is designed to create near-peak power consumption for processors. It uses assembly routines optimized by taking the specific microarchitecture of Fulhame into account. Firestarter tests the most important power consumers of compute nodes: CPU (cores + uncore compone-nents such as caches), GPUs, and main memory of Fulhame.

Firestarter was originally written for x86 instruction set architecture and the project had to be converted into AArch64 instruction set architecture so that it would run properly on ARM-based Fulhame. There is no direct conversion between two ISA's, but after benchmarking results come, rest can be tuned with the output from the HPC.

IOR-parser

When working with IOR, it outputs files with information about the system I/O performance with lots of data. When there are plenty of nodes and plenty of configurations, it is possible to automatise all the process of testing every possible configuration with a script, but how about the results?

IOR-parser, which repo is iorbench-tool, is the tool that allows in a automatised way, to create reports with charts about the performance of the system from its IOR results.

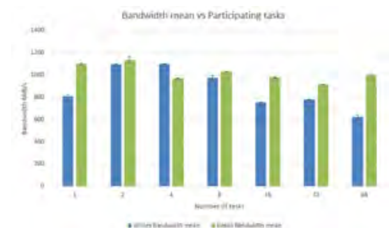


Figure 4: IOR report chart example.

IOR-parser can be also configured to get a customised report, with a different chart, or with more and different data. All is stated on the repo wiki.

```
//Initialize SSE-Registers for Transfer-Operations
"movabs $0x0F0F0F0F0F0F0F0F, %%x11;"
"pinsrq $0, %%x11, %%xmm14;"
"pinsrq $1, %%x11, %%xmm14;"
"shl $4, %%x11;"
"pinsrq $0, %%x11, %%xmm15;"
"pinsrq $1, %%x11, %%xmm15;"
"mov %%x0, %%x1;" // address for L1-buffer
"mov %%x0, %%x2;"
"add $23768, %%x2;" // address for L2-buffer
"mov %%x0, %%x3;"
"add $262144, %%x3;" // address for L3-buffer
"mov %%x0, %%x4;"
"add $33554432, %%x4;" // address for RAM-buffer
"movabs $8, %%x8;" // reset-counter for L2-buffer with 405 cache lines accessed per loop (202.5 KB)
"movabs $2542, %%x9;" // reset-counter for L3-buffer with 165 cache lines accessed per loop (26214.38 KB)
"movabs $3296421, %%x10;" // reset-counter for RAM-buffer with 40 cache lines accessed per loop (8241052.5 KB)
```

Figure 2: Firestarter example code of initializing x86 registers.

```
demangler on  master is  v1.0.0 via  v1.46.0
❯ ./target/x86_64-unknown-linux-gnu/release/dmgler --help
dmglr 1.0.0
Jerónimo Sánchez <jsg568@inlumine.ual.es>
Perform operations regarding mangled ARM cores when using OpenMPI.

USAGE:
    dmgler [OPTIONS] <SUBCOMMAND>

FLAGS:
    -h, --help       Prints help information
    -V, --version    Prints version information

OPTIONS:
    -m, --mpi-program <program>    Append 'program' to MPI recommended command. [default: <program>]
        --smt <smt>                SMT configuration (2 or 4). [default: 4]

SUBCOMMANDS:
    demangle    Demangle a set of Intel-styled threads
    get         Display a set of demangled threads that satisfy certain condition(s)
    help        Prints this message or the help of the given subcommand(s)
```

Figure 3: dmgler usage. This CLI allows an easy integration within an *slurm* script thus making this tool very useful in the already set pipeline of EPCC.

dmgler

dmgler is also CLI tool designed to solve a problem related to the use of OpenMPI on ARM systems.

The aforementioned problem consists in the following. When a program is wanted to be executed in a variety of computers, it needs some special code to share the operations of said program to the different computers of the cluster. **OpenMPI** is the *de facto* library for this type of task.

When the **OpenMPI** has to be initialised, it gathers the set of computers it can run on, and for each computer, the number of threads in which the program can be finally executed.

However, **OpenMPI** expects an Intel numbering style, in which every thread of each core from both sockets has an unique identifier on the whole computer blade, but the numbering it receives is the ARM one, which does not have an unique ID.

Table 1: Numbering style per architecture.

	4 Threads	
Core	Intel	ARM
1º	0-3	0,32,64,96
2º	4-7	1,33,65,97
3º	8-11	2,34,66,98

As shown, ARM does not follow the ascending ordering Intel has per core but spanned among the cores.

Once discovered which rules follows ARM on its ordering, a CLI tool was created. This tool, **dmgler**, is written in Rust, a systems programming language, focused on secure memory control without the hassle of mallo*cing* and free*ing*, and also without a *Garbage Collector*.

dmgler outputs a **OpenMPI** command line to run the desired threads on a ARM system with no need of *extra-thinking*. The tool is used as Figure in 3.

The primary utility of **dmgler** is to be useful in an already existing environment, being said environment *slurm*.

PRACE SoHPCProject Title
Porting and benchmarking on a fully ARM based cluster

PRACE SoHPCSite
EPCC, Edinburgh, Scotland

PRACE SoHPCAuthors
Irem Kaya and Jerónimo Sánchez, Turkey and Spain.

PRACE SoHPCMentor
Nick Johnson, EPCC, Scotland

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

PRACE SoHPCSoftware applied
Visual Studio Code

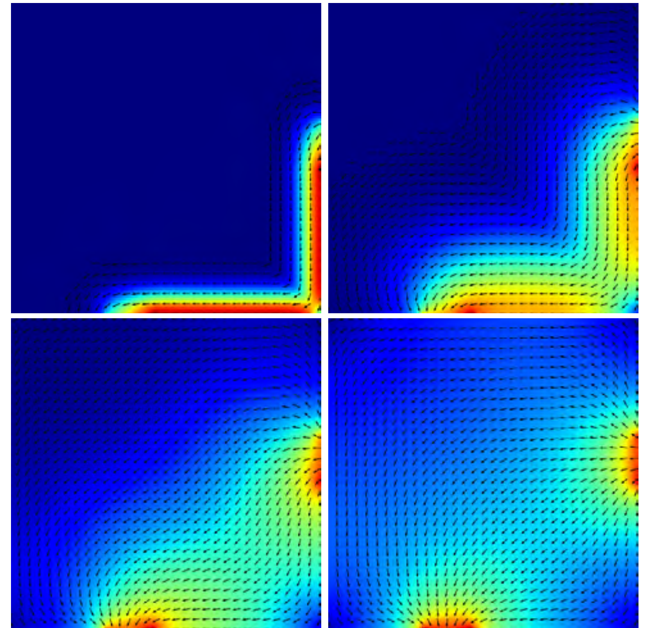
PRACE SoHPCMore Information
Visual Studio Code

PRACE SoHPCAcknowledgement
We thank Phil Ridley from ARM for his advice and input to this project.

PRACE SoHPCProject ID
2006

How to make Python code run faster

Alexander J. Pfleger
Antonios-Kyrillos Chatzimichail



There are several techniques that can be applied to speed up Python codes. The goal of this project is to investigate optimisations for Python programs that run not only on CPUs but also on GPUs.

Nowadays, the use of Python is getting popular, mainly because it's user-friendly and saves quite a lot of developing and debugging time. In this project a short Python programme is investigated. The program performs a Computational Fluid Dynamics (CFD) simulation of fluid flow in a cavity. The Fluid Dynamics problem is a continuous system that can be described by the partial differential equations $\nabla^2\psi = 0$. In order for a computer to run simulations, the calculations need to be put into a grid (discretisation). In this way, the solution can be approached by finite difference method, which means that the value of each point in the grid is updated using the values of neighbouring points. The update scheme can be seen in fig. 1.

The program can be parameterised by specifying the variables below:

Scale Factor (`sf`) affects the dimensions of the box cavity and consequently the size of the array(s) in which the grid is stored.

Number of Iterations affects the number of the steps in the algorithm, the larger it is the more accurate the result will be.

Reynolds number (`Re`) defines the viscosity which affects the presence of vertices (whirlpools) in the flow.

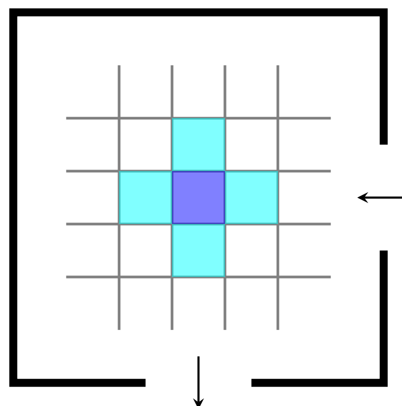


Figure 1: The blue point of the grid is updated using the cyan top, bottom, left and right points.

The simulation result is visualised by arrows and colours drawn in an image representing the grid. The arrows demonstrate the direction of the fluid at each point, while the different colours indicate the fluid's speed, with blue being low speed and red being high speed. The title picture of this report shows the result of a simulation with `Re = 0` and `sf = 4`. The pictures were exported after different numbers of iterations (`50`, `500`, `2 000`, `100 000`).

As a baseline serial code, we selected the fastest Python program that was developed by last year's student, Ebru Diler. The program uses the Numpy module to make fast array calculations. But it turns out that there are several ways to further optimise the existing serial code.

CPU optimisations

The following methods are directly applied on (excerpts of) the baseline code. They describe general concepts and can be easily adapted to any other Python project.

Choosing the right implementation

For the first method, the performance of some built-in functions are compared. We will discuss this issue on the basis of the square function A^2. In the project it

is used in the distance function:

$$d_{p,q} = \sqrt{\sum_i (p_i - q_i)^2}$$

It calculates a scalar $d_{p,q}$ from the equally sized matrices p and q.The distance function quantifies the difference between the two matrices. It is used to stop the iterative process when the output is merely changing and therefore a minimum is reached. In this example four different implementations for the expression A^2 are looked at:

- `for i in range(m): A[i]*A[i]`
- `numpy.power(A,2)`
- `A**2`
- `A*A`

A random NumPy array is created with `A = numpy.random.rand(m)`, where `m` denotes the size of the array.

The four different implementations for the square function A^2 were timed for different matrix sizes $\texttt{m} \in [2^2, 2^{26}]$. The results can be seen in fig. 2. Unsurprisingly the `for`-loop is rather slow. The simple multiplication may be up to one-hundred times faster. The `numpy.power()` function is somewhere between. This may be more of a surprise. The `numpy.power()` function may be written especially for NumPy arrays. Of course, it provides more options and *is* faster for larger exponents (> 70). For simple expressions like the square, an inline implementation should be chosen.

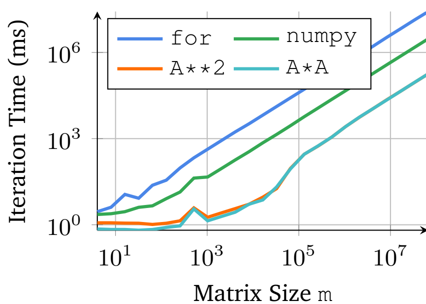


Figure 2: Execution time over matrix size `m` for different implementations of A^2.

Python Binding - C inside of Python

The second method is about Python bindings. In a Python binding, C-code is called from Python. There are several different libraries to achieve this. One of the most popular libraries is pybind11. It is rather compact and requires only a few additions to existing C-code. Also, many marvellous C++ features can be used, since pybind11 uses a C++ compiler.

Again, we will use the square function A^2 to explain this method. Firstly, we use only single values for A. In C++, the function can be written like this:

```
int square(int A){ return A*A;}
```

In order to use this function in Python, it needs to be converted to a Python module. This can be done by altering the code like followed:

```
#include <pybind11/pybind11.h>

int square(int i){ return i*i;}

PYBIND11_MODULE(bind_sq,m){#
    m.def("squareCPP", &square,
    "NOTE: squares integers");}
```

In the first line, the pybind11 library is included in C++. The second line is the already implemented square function. The last lines generate the actual module. Also, short documentation can be added. After compiling the C++ code, the module can be loaded and used in Python:

```
from bind_sq import squareCPP
squareCPP(A)
```

To use more advanced functions, the same concepts need to be applied. To use NumPy arrays like in the previous example, some further additions need to be made in the C++ code. About three lines need to be added for each array. Those cases are well explained in the pybind11 documentation.[1]

The performance boost of pybind11 using arrays is shown in fig. 3. For this figure, the jacobi function is used, since it has more impact on the project. The new module is ten times faster than the already optimised Python code. The performance is similar to a stand-alone C++ program. The third line in this plot is generated by a parallelised module and provides a second boost by a factor of ten. We will have a look at this method in the next paragraphs.

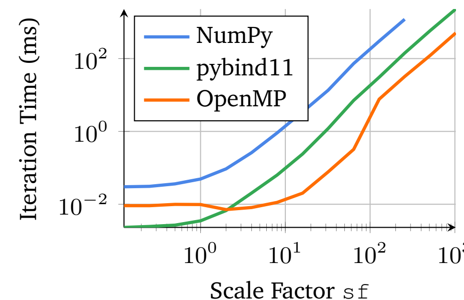


Figure 3: Execution time over `sf` for different implementations of the jacobi function.

Parallelised Python Binding

To further increase the performance of the function, parallelisation techniques like OpenMP can be used. The C++ code has to be slightly altered but no changes are made in the python project. This helps to keep the code clean while parallelisation is done in the background. In fig. 3 the performance of an OpenMP module with 18 threads is compared to the performance of the simple pybind11 module. Depending on the problem size a drastic speed-up can be noticed.

Still, OpenMP holds some pitfalls. As we investigated on the HPC-system Cirrus, the performance is unpredictable above 18 threads, sometimes dropping and sometimes increasing up to 36. In fig. 4, the performance is plotted over the number of used threads, for eleven runs. For each run, the transition from the upper-performance curve to the lower performance curve takes place somewhere between 18 and 36 threads.

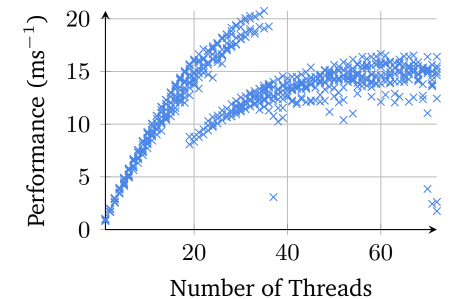


Figure 4: Performance over number of threads of the Pybind11/OpenMP module for the jacobi function. Eleven runs for a problem with `sf = 32` are compared.

The significance of 18 is that it is the number of physical CPU cores (ignoring hyper-threading) in a CPU - each node has 2 CPUs. The performance issues could be due to NUMA effects - whether or not the memory is allocated on the same CPU as the thread that accesses it.

Numexpr

After some research, we found out that there is a Python module, named Numexpr,[2] which performs better than Numpy, mainly because it produces less temporary arrays when evaluating numerical expressions. Apart from that, it uses multithreading internally, enabling parallelisation for the calculations that further boosts the performance. On fig. 5, it can be seen that the Numexpr version performs 3 times faster than the baseline Python program, even when

using one single thread, while it outperforms the serial C program as more threads are used.

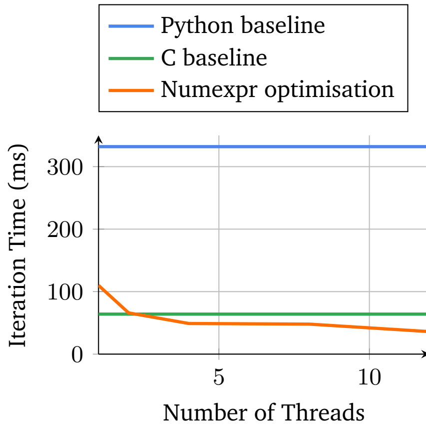


Figure 5: Iteration Time over Number of Threads. All tests ran on Archer with `Re = 2` and `sf = 64`.

Parallel Numexpr

Numexpr module also provides a significant performance boost to the preexisting MPI Python program. Of course, the performance depends on the number of MPI processes and the number of threads for each process. The total number of threads can be found by multiplying MPI processes by threads per process. A single node on the Archer supercomputer has 24 cores, thus can have up to 24 threads, one running on each core. By all means, there are many possible combinations of processes and threads to fully use a single node. Some of them can be spotted on tab. 1, where their performance is compared. After running tests on both x86-based Archer and ARM-based Fulhame supercomputers, it seems that the best option is to use as many single threaded MPI processes as possible within a compute node. This conclusion is crucial because it reveals what is the best way for our case to use a node of a supercomputer.

Table 1: Iteration Time (ms) for different number of MPI processes and threads.
All tests ran on Archer with `Re = 2` and `sf = 72`.

Python MPI with Numexpr and # Threads:						
# MPI processes	1	2	3	6	12	24
1	145	91	75	59	48.3	49.7
2	57	37	31	25.48	25.44	
4	31	21	18	16.6		
8	18	13	12.6			
12	15	11.0				
24	10.7					

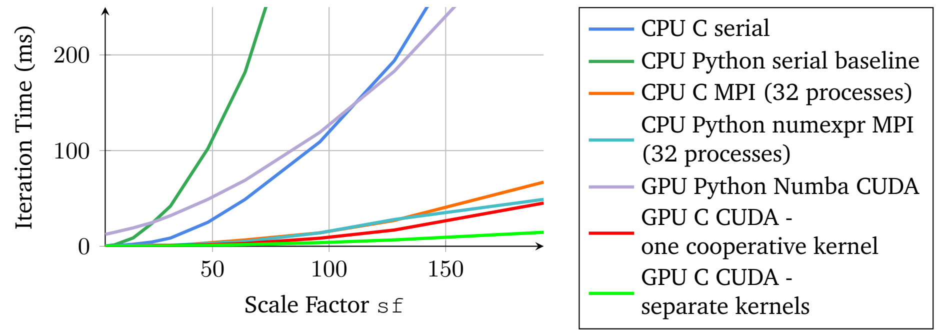


Figure 6: Iteration Time over `sf`. All tests ran on Cirrus with `Re = 2`.

GPU program implementation

In recent years, there is a trend to use GPUs not only for gaming but also for general purpose, especially for programs that make few decisions but do many calculations. That's a logical thought to make if you consider that instead of running tens of parallel MPI processes on the CPU, the GPU offers the ability to run thousands of threads in parallel. In our case, the optimal is to assign every point of the matrix to a separate GPU thread. The whole matrix is stored in the GPU's memory, so any thread has direct access to any point of the matrix without exchanging any messages like MPI.

GPU programs were developed for Python and C and the performance tests ran on a novel HPC system, Cirrus. The Python GPU program, which is implemented with the Numba CUDA module, can not compete with the much faster CPU parallel code, but it surpasses the serial codes, as fig. 6 shows. On the other hand, two versions of CUDA C were developed and both outperformed the parallel codes. One launches multiple kernels (functions executed on the GPU) per iteration, and the other launches a single cooperative kernel for all iterations. Someone could predict that the latter version would be faster, because it avoids the overhead of launching many kernels. However, this overhead is minor and because of the fact that the second version is using the CUDA runtime launch API which applies some limitations to the amount of GPU blocks, it is quite faster to launch multiple smaller kernels and therefore the first version is the most optimal.

Conclusion

In this article, we have described a number of ways to improve the performance of python codes.

One surprising result is that the parallel Python program outperforms the C parallel one. This is hard to explain, but maybe the reason behind this lies in the implementation of the Numexpr module. Perhaps, also, the C parallel code is not fully optimised. Further investigation should be done in the future regarding first-touch memory techniques in pybind11/OpenMP modules to obtain more predictable behaviour. Additionally, the Numba CUDA code could be revised, because its performance is worse than expected. However, the GPU program using CUDA C drastically outperforms the C code, by achieving 30 times better performance.

References

[1] Readthedocs pybind11 documentation. https://pybind11.readthedocs.io

[2] Numexpr module https://github.com/pydata/numexpr

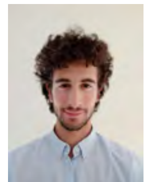
PRACE SoHPC Project Title
Performance of Parallel Python Programs on New HPC Architectures

Alexander J. Pfleger

PRACE SoHPC Site
Edinburgh Parallel Computing Centre (EPCC), UK

PRACE SoHPC Authors
Alexander J. Pfleger, Graz University of Technology, Austria
Antonios-Kyrillos Chatzimichail, University of Thessaly, Greece

PRACE SoHPC Mentor
Dr David Henty, EPCC, UK



Antonios-Kyrillos Chatzimichail

PRACE SoHPC Contact
Alexander J. Pfleger, Graz University of Technology
E-mail: pfleger@plancks.at
Antonios-Kyrillos Chatzimichail, University of Thessaly
E-mail: antonis.xatzimixail@gmail.com

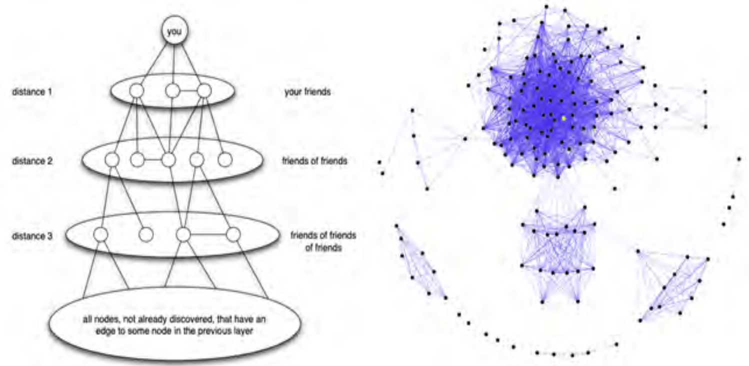
PRACE SoHPC
d
Python, NumPy, pybind11, Numexpr, Numba

PRACE SoHPC Acknowledgement
Great thanks to Dr. David Henty for continuous support and guidance throughout the whole project.

PRACE SoHPC Project ID
2007

BFS Graph Traversal with CUDA

Berker Demirel



The motivation of the project is implementing Breadth-First Search algorithm with achieving high parallelism. We achieved to perform competing results with Gunrock library that can serve fast Social Network traversal.

Introduction

Together with the abundance of data in recent years, the size of many interesting real-world and scientific problems that have been modeled as graphs is drastically increased. Although machines today can store the data in their memory with no problems, efforts to improve the performance of algorithms running on these graphs have only recently displayed a positive acceleration. In the last decade, the interest in parallelizing the graph processing has resulted in remarkable works such as Ligra [1] and Gunrock [2], which offer parallel solutions to many graph problems.

In this project, we focus on the Breadth-First Search(BFS), which is a well-known graph traversal algorithm. Given a graph and a source node, BFS visits every vertex and edge in a well-defined order and assigns a distance to every vertex, corresponding to the length of the shortest path from the source to the vertex.

In addition to the fact that it is used as a sub-procedure in many graph algorithms like finding strongly connected components or checking if a graph is bipartite, BFS is important because it is directly used in real-world problems such as Social Network Applications. Graph500 Benchmark [3] is followed to determine input format and graph generation since it offers Kronecker Graph generation to model Social Networks.

Methods

Graph Dataset & Representation

We used Graph500's Kronecker graph generator code to generate our graphs. Given graph scale -which is the logarithm base two of the number of vertices- and edge factor -edge ratio of the number of nodes-, it generates a text file of the graph as an edge list.

Since we are dealing with very large graphs, it is needed to decide our graph storage format(representation) carefully considering memory issues. Rather than using classical adjacency matrix representation using $O(V^2)$ memory, it is more suitable to use Compressed Row Storage(CSR) format with $O(V+E)$ memory due to the fact that we are dealing with sparse graphs whose edge factor are significantly smaller than their number of nodes.

Different BFS Approaches

There are three different novel approaches of BFS: top-down, bottom-up and hybrid. In the top-down approach, the algorithm tries to reach an unvisited node from a visited node by investigating the visited node's neighbors. On the other hand in the bottom-up approach, the algorithm tries to reach a visited node from an unvisited node by iterating over unvisited node's adjacents. Finally, the hybrid method combines these two different novel approaches. It starts with a top-down approach and if the number of edges of visited nodes exceeds some threshold (like $1/8$ of all edges), it switches to bottom-up approach in order to skip all the edge checks from the large frontier.

Implementation Details

We implemented 6 different approaches for BFS.

	CPU	SBFS	QBFS	CPU_Q GPU_S	GPU_S GPU_S	GPU_Q GPU_S
graph_20_16	0.28	1.24	0.98	**15.59**	4.11	3.11
graph_21_16	0.25	1.41	1.10	**16.61**	5.31	4.20
graph_22_16	0.25	1.35	1.28	**18.48**	7.44	6.36
graph_23_16	0.19	1.55	1.51	**18.95**	8.51	7.26
graph_24_16	0.16	1.92	1.79	**20.03**	12.23	10.74
graph_25_16	0.13	2.16	2.03	**20.52**	11.97	10.54
graph_26_16	0.12	2.23	2.33	**22.61**	14.72	13.32

Figure 1: GTEPS values of all GPU implementations on generated graphs (graph name format: graph_SCALE_EDGE_FACTOR)

SBFS: It is a top-down approach that scans the set of vertices at each iteration to determine the current frontier. GPU implementation does not have any atomic operations that reduce parallelism.

QBFS: It is a top-down approach that utilizes a FIFO queue. It iterates over the nodes in the frontier and checks if any of the adjacents is unvisited and adds into the next queue atomically.

BUBFS: It is a bottom-up approach that uses the same technique as SBFS except for the direction.

OMP-Q–GPU-S: It is a hybrid method that combines OpenMP Queue top-down approach with GPU-BUBFS scanning bottom-up approach.

GPU-S–GPU-S: It is a hybrid method that uses scan BFS for both top-down and bottom-up approaches.

GPU-Q–GPU-S: It is a hybrid method that uses QBFS as a top-down and BUBFS as a bottom-up approach.

Experimental Results

The experiments were performed on ICHEC's primary supercomputer Kay. It has a GPU partition of 16 nodes. On each node, there are 2xNVIDIA Tesla V100 16GB PCIe (Volta architecture) GPUs each having 5,120 CUDA cores and 640 Tensor Cores. The code was written in C/C++, CUDA and compiled using GCC version 8.2.0 and NVCC version 10.1.243 with the optimization flag -O3 enabled (publicly available on gitlab) We generated 7 graphs from scale 20 to 26 and in order to evaluate the performance of the implementations, we randomly choose 64 connected sources to start the search.

The GTEPS(Giga traversed edges per second) results of our algorithms are as in Figure 1 on single GPU. It can be seen from the results that OMP-Q–GPU-S implementation achieved the best performance on all of the graphs. It is even two times faster than its succeeding opponent and performs up to x188.42 speedup against the sequential CPU implementation.

All of the implementations are controlled with CUDA-memcheck(for memory leaks) and profiled with nvprof. Then, applied kernel analysis with NVIDIA Nsight Compute to investigate cache hits, speed of light statistics, etc. Since graphs are highly unstructured, our cache hit ratio does not depend on our implementation but depends on the input graph's topology.

After observing the best implementation, we wanted to compare it with an NVIDIA supported library, Gunrock. The GTEPS results of OMP-Q–GPU-S versus Gunrock's Direction Optimizing BFS(DOBFS) are illustrated in Figure 2.

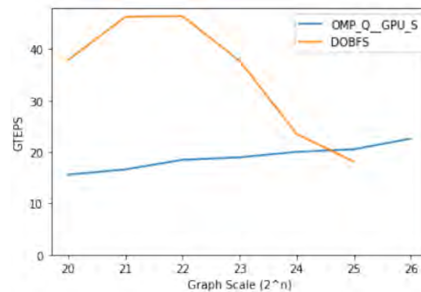


Figure 2: GTEPS values of Gunrock-DOBFS and OMP-Q–GPU-S versus graph scale

Although Gunrock's results dominate in smaller scales, we observe that its performance drops drastically especially after scale 22. On the other hand, OMP-Q–GPU-S increases its GTEPS slowly but surely with larger graphs which are supporting the fact that our implementation is more scalable than Gunrock's DOBFS on single GPU. Another point that might be stressed is, for graph scale 26, OMP-Q–GPU-S increases its speed at the usual rate while Gunrock's implementation gives out of memory error. It, therefore, shows us that for a single-GPU graph applications our implementation is using less memory compared to DOBFS. One can claim that Gunrock is not designed for single-GPU graph applications, however, it does not mean these performances are insignificant.

Conclusion and Future Work

As a result, we achieved to implement a successful GPU implementation of BFS that especially works well on social networks. The strength of our implementation is that it even competes with Gunrock Library however weaknesses such as processing larger graphs are left to future work. Dealing with larger graphs might be handled with partitioning graph into GPU memory. Unified Virtual Memory or a multi-node GPU implementation that contains a communication framework(like MPI) are possible candidates that can solve the issue.

References

[1] Julian Shun and Guy E. Blelloch (2013). Ligra: A lightweight graph processing framework for shared memory.

[2] Gunrock (2019). NVIDIA Supported CUDA Graph Library.

[3] Graph500 (2020). Supercomputer rating list for data intensive applications.

[4] S. Beamer, K. Asanovic and D. Patterson (2012). Direction-optimizing Breadth-First Search.

PRACE SoHPCProject Title
GPU acceleration of Breadth First Search algorithm in applications of Social Networks

PRACE SoHPCSite
ICHEC, Ireland

Berker Demirel

PRACE SoHPCBerker Demirel
Berker Demirel, Sabanci University, Turkey

PRACE SoHPCMentor
Buket Benek Gursoy, ICHEC, Ireland

PRACE SoHPCContact
Berker, Demirel, Sabanci University
Phone: +90 530 912 3365
E-mail: berkerdemirel@sabanciuniv.edu

PRACE SoHPCSoftware applied
C/C++, CUDA, OpenMP, Gunrock, Nsight Compute

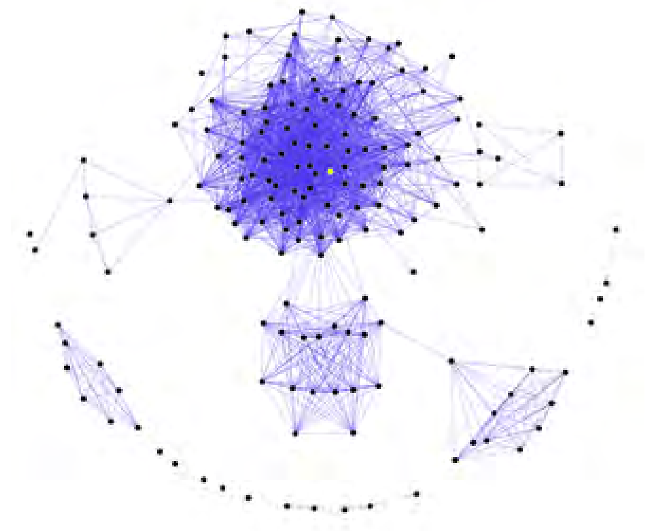
PRACE SoHPCMore Information
https://summerofhpc.prace-ri.eu/author/berkerd/

PRACE SoHPCAcknowledgement
Special thanks to Buket Benek Gursoy, Busenur Aktilav and those who have supported me furthering my research experience through feedback.

PRACE SoHPCProject ID
2008

GPU Acceleration of BFS

Busenur Aktılav



A Breadth-First Search (BFS) is one of the core graph-based searching algorithms. In this study, parallel implementation of BFS on GPU using CUDA is implemented and performance analyses are examined.

Large scale graphs are widely used in representing many practical applications. As graph domains are growing in size, the need for massively, parallel hardware like the GPU arises. In recent years, GPUs have become widely used for accelerating many codes because of their high computational power, good energy efficiency, and low cost. [1] Therefore, we utilised GPU to speed up the graph processing.

A Breadth-First Search (BFS) is one of the core graph-based searching algorithms. It is used as a building block for many higher-level graph analysis algorithms. That makes it very suitable for social network analysis. There are many applications to social networks such as node similarity, community detection, influential user detection. However, parallel implementation of BFS is very challenging due to irregular memory access and unstructured nature of the large graphs.

BFS is used in different benchmarks. One of them is Graph500 benchmark [2] which is based on a BFS in a large undirected graph. Throughout the research, Graph500 benchmark is taken as a pattern.

- Graphs are generated by Kronecker generator and they are represented.
- Parallel BFS search of some random vertices is achieved (64 search iterations per run)

TEPS (traversed edges per second) performance metric is used. In this research, the implementation of serial BFS algorithm on CPU and parallel BFS algorithm on the NVIDIA GPUs using the CUDA model is presented.

Datasets & Graph Representation

Kronecker graph generator in Graph500 routine is used to generate graphs. By giving the scale and the edge factor, it generates a txt file consisting of edges. Scale determines the number of vertices, edge factor determines the number of edges in the generated graph.

There are two different graph representation technique: Dynamic and static. If the vertices and edges are changing, then dynamic data type should be used for representation. If the size of the data type is determined before the execution of the program, then static data type could be used. In this study both dynamic and static data types are used representing the graphs.

Table 1: Static-Dynamic CPU performance

scale	static (ms)	dynamic (ms)
20	114.3	672.1
21	264.2	1430.9
22	607.2	2970.9
23	1435.06	7128.9
24	3383.1	16659.1

Table 1 shows the serial BFS algorithm performance analysis. Static representation has shown better performance than the dynamic representation.

BFS Algorithm and cuBFS

There are two different approaches for BFS algorithm: Top-down and Bottom-up. In this research, a top-down BFS algorithm using a queue is implemented and it is referenced as serial BFS for CPU and cuBFS for GPU implementation.

Consider graphs of the form $G = (V, E)$ with the set V of n vertices and a

set E of m edges. Given a source vertex v_s, the goal is to traverse the vertices of G in breadth-first order starting at v_s. Each newly discovered vertex v_i will be labelled by its distance d_i from v_s and the predecessor vertex p_i immediately preceding it on the shortest path. It performs linear $O(m + n)$ work.

The FIFO ordering of the serial algorithm labels vertices in increasing order of depth. The idea of parallel BFS algorithms is to process each depth level in parallel. Algorithm 1 illustrates this approach. Its work complexity is $O(n^2+m)$ [3]

Algorithm 1: Parallel cuBFS

```
parallel for (i in V):
  distance[i] := ∞
distance[s] := 0
iteration := 0
do
  done := true()
  parallel for (i in V):
    if (distance[i] == iteration)
      done := false
      for (offset in R[i]..R[i+1]-1:)
        j := C[offset]
        distance[j] = iteration+1
  iteration++
while (!done)
```

Experimental Results

All experiments are conducted on KAY supercomputer. It comprised of a number of components. Its thin component has 336 nodes (2x20cores) Intel Xeon Gold 6148, 2.4Ghz, 192GiB RAM 400 GiB SSD and its GPU component has 16 nodes with the same specification as the thin component, with the addition of 2xNVIDIA Tesla V100 GPUs on each node, Each GPU has 5120 CUDA cores and 640 tensor cores. The serial algorithm is written in C/C++ language and the parallel algorithm is written in CUDA. For compilation gcc version 8.2.0 and nvcc version 10.1.243 is used.

Table 2 & 3 shows summaries of the experimental result of CPU and GPU BFS algorithms with static and dynamic representation. As you can see that we achieved speed-up in parallel implementation so cuBFS is faster than the serial BFS algorithm in both representation techniques. Also notice that while scale is growing, the parallel cuBFS GTEPS is constantly increasing. On the other hand, serial BFS performance is decreasing for larger graphs.

We compared our results with Gunrock [4] which is a CUDA library for graph-processing designed specifically for the GPU. It has a better performance compared to our results.

Table 2: Static BFS Algorithms(GTEPS)

scale	serial BFS	parallel cuBFS
20	0.27	5.6
21	0.24	9.66
22	0.21	10.08
23	0.18	13.2
24	0.15	13.3

Table 3: Dynamic BFS Algorithms(GTEPS)

scale	serial BFS	parallel cuBFS
20	0.049	0.59
21	0.046	0.67
22	0.045	0.78
23	0.037	0.95
24	0.032	1.12

Code Profiling

The different code profiling tools are used to analyse the the GPU code. These tools are nvprof, nvvp, NVIDIA Nsight Compute.

- **nvprof** creates detailed profiles of where codes are spending time and what resources they are using.
- **nvvp** collects and analyzes the low-level GPU profiler output for the user and It gives timeline analysis of the running code.
- **NVIDIA Nsight Compute** [5] is an interactive kernel profiler for CUDA applications. It gives detailed workload analysis, different statistics and occupancy of hardware.

After analysing the parallel implementation of BFS algorithm by profiling tools, several challenges are observed. Warps are not used efficiently because of irregular workloads. There is a load imbalance issue meaning that the computation on each iteration differs greatly. In the case of low computation, the efficiency decreases because GPU is underutilised. The arbitrary references from each thread within the warp result in poor coalescing. Thus, it decreases the performance of the code. We worked on single GPU and large scale graphs are not examined due to memory constraint on GPU.

Conclusion and Future Work

As a result, We achieved to implement serial BFS on CPU and parallel BFS on GPU using CUDA. Performance analyses of codes are made. It's seen that the parallel BFS is much faster than the serial code. Different code profiling tools are used to have a better understanding of GPU kernel activity and timeline analysis. The weaknesses of GPU code are detected and they are as follows: warps inefficiency, load imbalance issue and failure of analysing larger scale graphs.

In future, algorithm could be enhanced to use warps efficiently. Also, larger scale graphs could be handled by graph partitioning and multi-node GPU could be used.

References

[1] D. Tödling, M. Winter and M. Steinberger, "Breadth-First Search on Dynamic Graphs using Dynamic Parallelism on the GPU," 2019 IEEE High Performance Extreme Computing Conference (HPEC), Waltham, MA, USA, 2019, pp. 1-7, doi: 10.1109/HPEC.2019.8916476.

[2] Graph500 (2020). Supercomputer rating list for data intensive applications.

[3] Duane Merrill, Michael Garland, and Andrew Grimshaw. 2015. High-Performance and Scalable GPU Graph Traversal. ACM Trans. Parallel Comput. 1, 2, Article 14 (January 2015), 30 pages. DOI:https://doi.org/10.1145/2717511

[4] Gunrock (2019). NVIDIA Supported CUDA Graph Library.

[5] NVIDIA Nsight Compute (2019.5). interactive kernel profiler for CUDA.

PRACE SoHPCProject Title
GPU acceleration of Breadth-First Search algorithm in applications of social networks

Busenur Aktılav

PRACE SoHPCSite
Irish Center for High-End Computing, Dublin, Ireland

PRACE SoHPCAuthors
Busenur Aktılav, Izmir Institute of Technology, Turkey

PRACE SoHPCMentor
Buket Benek Gursoy
ICHEC, Ireland

PRACE SoHPCContact
Busenur Aktılav
Izmir Institute of Technology
E-mail: busenuraktilav@gmail.com

PRACE SoHPCSoftware applied
C/C++, CUDA, Nsight Compute

PRACE SoHPCMore Information
https://summerofhpc.prace-ri.eu/author/BusenurA/

PRACE SoHPCAcknowledgement
I am deeply thankful to my project mentor Buket Benek Gursoy and my teammate Berker Demirel for their supports. Many thanks to Assoc.Prof.Dr. Leon Kos for his helps. Special thanks to Assist.Prof.Dr. Işıl Öz for her supports.

PRACE SoHPCProject ID
2008

From the data to the field, using deep neural networks to solve astrophysical problems and then applying the solutions to edge devices.

Deep Neural Networks for galaxy orientation.

Andrés vicente

We made a machine learning model to improve and automatize the manual and analytical techniques of computing galaxy orientations. We try different model architectures, implemented a custom loss function, and obtained promising results. We then prove that we can run the model on a system with very constrained resources.

The objective of the project is to use Deep Neural Networks (DNN) to detect objects in images and we are lucky because, in the Astrophysics world, a very big portion of the data obtained from the Universe is in form of pictures. In the past, most of the classification of the galaxy morphologies was done by simple human inspection. Nowadays, we have better tools to classify the galaxies but almost all of them need to be applied using an analytical model and fit it to every one of the observations made, which is obviously time-consuming and computationally expensive. DNNs and his object detection capabilities open a new world of possibilities in Astronomy and Astrophysics because we will not only be able to detect morphologies of galaxies (which is a rather easy task), but we also could go beyond that and infer physical properties of the galaxy just by looking at the raw image!

In our case, we will try to detect the orientation of galaxies (which is closely related to the angular momentum vector if you are wondering). This is not an easy task since we don't have "training data" to feed our network because we don't know the ground truth of this magnitude in the observed galaxies but... Here comes the HPC again to rescue us.

We can mock the observed data with high-resolution simulations of galaxies were we know all the parameters. These simulations are done by experts in the field in the most powerful supercomputers in the world for example the Illustris simulation done at PRACE supercomputers: https://prace-ri.eu/universe-simulation-illustris-is-an-ongoing-success-story/.

We decided to use the NIHAO[1] simulation because it has 100 high-resolution galaxies with different morphologies.

From these simulations, we can render images of galaxies in any orientation and point the angular momentum as a vector as we can see in the image at the top.

These physical properties of the galaxies are relevant because they tell us the story of the galaxy evolution and how it has been formed and evolved. This helps us to understand our galaxy and somehow why the Universe is as beautiful as it looks.

We used a convolutional deep neural network to do the job of detect the orientation in the rendered data. We tried different architectures but the conceptual scheme (simplifying a lot) will look something like Figure 1.

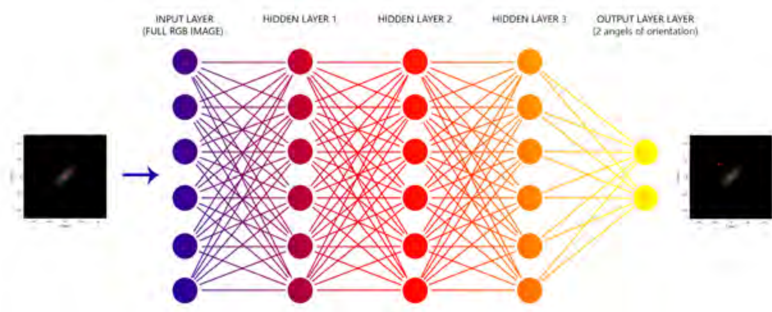


Figure 1: Conceptual image of a DNN that takes a raw images and predicts the orientation.

The process

The first we needed to make to have our network is to create the dataset to train the network with. For that, we used the NIHAO simulation and the pybody python package. This package allowed us to center and rotate the galaxy to obtain, for each galaxy in the simulation, around 10 images of the different perspectives, obtaining a total of around 900 images. All the images were computed with his corresponding labels that consisted of the 3D vectors of the total angular momentum of the galaxy to latter train the network.

When we had the data, we continued by making the pipeline and adapting the existing model to work with our data. We made a pipeline to filter the data in order to remove the smallest galaxies and to convert the labels from 3D to 2D, projecting the 3D vectors into the image plane. We also implemented dataset augmentation by performing random flips in both axes and random rotations of 15 degrees. With this, we ensure that in each step of the training, the network does not see the same data and therefore is more difficult to overfit the model. Overfitting means that the network memorizes the images and associates them to a particular output but does not generalize the knowledge to apply it to new images.

The model was made by adapting 2 existing models of convolutional neural networks (CNNs) (ALEXNET[3] and RESNET[2]). CNNs works by applying a series of convolutions or filters to the image, extracting in each layer more high-level information to end up with a general knowledge of what features make galaxies have a particular orientation. In order to work with our images and also to give numerical outputs and not categorical ones, as is common in these kind of architectures that are made to classify images, we extended the models by customising the last layers.

Now that we know what architecture to use, how we are going to prepare our data, and how to visualize the results, it seems that is almost done right? Well ... of course it is not that simple. One key aspect of training a network is defining what is wrong and what is right and in a classification model, it's a rather easy task since we have categories between which we need to choose. In a model like ours, in which we need to predict a continuous parameter (in fact 2 in this case), things get a bit trickier.

We need to make a function that tells not if we are wrong or not, but by how much we are wrong. This gets even worse since we have multiple solutions for the orientation of a galaxy as the direction of the angular momentum vector can go in both directions of the orientation (axis of rotation) of the galaxy as shown in Figure 3. How can we define how far is our model from reality then?

The approach we chose, as we are using vectors, is to use and adapt the existing Cosine Similarity loss function. It's a measure of the angular distance of the two vectors but taking into account that a $+180^{o}$ vector is a valid solution. It is computed with the dot product and the magnitude of the vectors A and B as:

$$\text{Custom loss} = -\left|\frac{\mathbf{A}\cdot\mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|}\right|$$

This approach has an output between -1 and 0 where -1 is a perfect score (output in line with the axis of rotation in either direction) and 0 is totally off prediction (90° offset), but we need to take into account that it is not a linear loss so the offset angle of our predictions will decrease as the $\propto arcos(loss)$.

With that, we started the training and testing process in our machines, with everything installed in a Docker container. This setup very quickly became infeasible for the amount of data (even with a very reduced dataset) due to compute times on a normal PC. We migrated our environment to the Salomon cluster hosted at IT4Innovations, the National Supercomputing Center of Czech Republic. For running the models at the cluster, we needed to export our configuration in the Docker container with all the modules and libraries as well as all the files and data to their machine. When we had the first running prototypes of the model on the CPU-only Salomon cluster, we migrated to the Barbora cluster which holds four NVIDIA Volta 100 GPUs per node. This accelerated the training of the CNNs (significantly) due to the highly parallelization nature of the underlying hardware. The speedup we obtained with this was about 10 to 20 times, depending on the selected hyperparameters. That allowed us to make much more experiments as we can see in figure 2 where we show the loss function described above for some of the runs that we made to test and fine-tune the models.

As a culmination of the project, we exported the model to run on an edge device to prove that this model can be applied on the edge. One example application could be in the next generation of robotic telescopes. Where we need to detect the orientation of the galaxy to put the slit in order to automatize the process of obtaining galaxy rotation curves. We chose a Raspberry Pi for this task as is the most resource-constrained device we can think of that also is very popular for Edge applications.

The Results

The results of most of our runs are illustrated in Figure 2 where we can see the loss of the validation and training in the different epochs of the training process. We can see that we had a gap between the training and the validations datasets. That is because the training dataset is what the network is optimised for and the validation dataset is data the network is tested with. Despite it is usual to have a gap, we found an unusually large difference that we manage to correct with the augmentation of the dataset. We also can see that the validation loss is around -0.9 to -1, which means almost a perfect score, however, the validation is around -0.8 being the best we achieve -0.85. A table with the results from the different networks is provided.

Table 1: Random table

Name	Losses	
CNN	Train	Validation
ResNet18	−0.87	−0.854
ResNet34	−0.876	−0.852
ResNet50	−0.908	−0.867
resnet cust.	−0.803	−0.793
Alexnet	−0.841	−0.823
custom	−0.884	−0.851

Regarding the application to an edge device, the inference time was of about 5 to 10 seconds per image depending on the model. This is a perfectly valid time since the acquisition times for these kinds of images in real telescopes can be

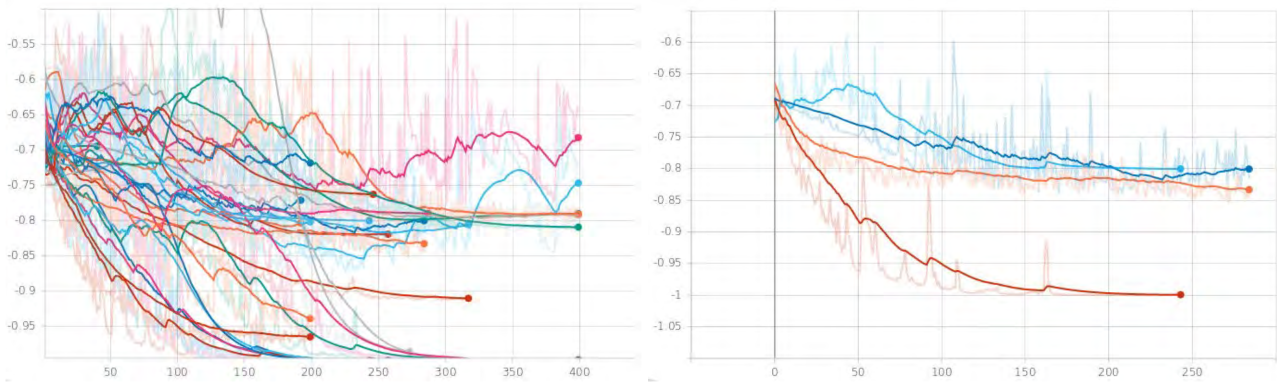


Figure 2: at the left we can see some of the training runs with the gap between the plateau in validation and training losses. At the right we can see two of those runs for a resnet18 model before augmentation (validation is light blue and training red) and after (validation dark blue and training orange), where the gap between training and validation was solved.

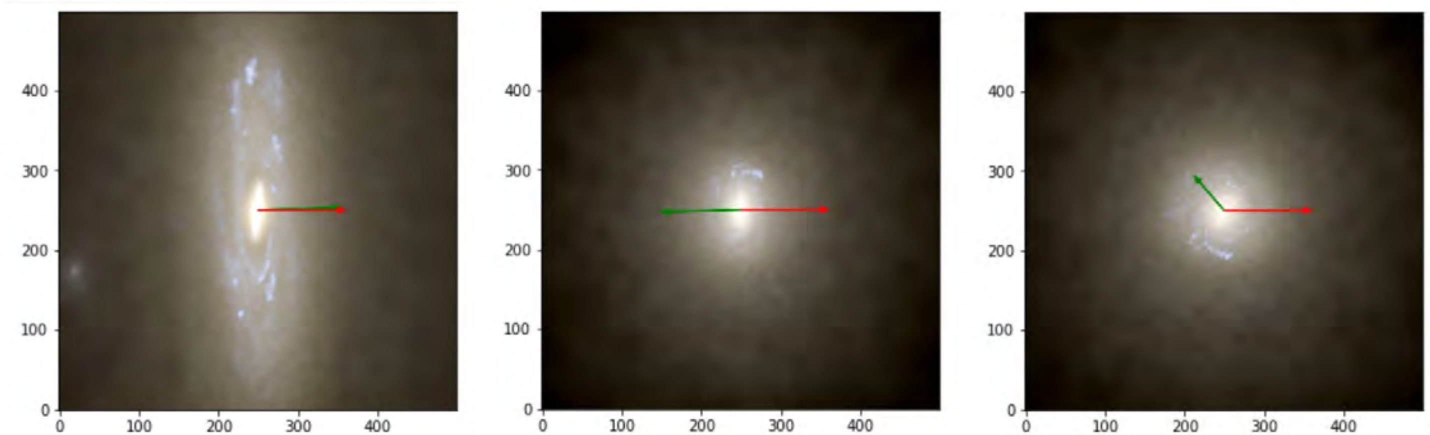


Figure3: We can see different predictions (in red) and labels (in green) for 3 of the galaxies in our validation dataset. Notice that the middle image is a good prediction since we take into account 180° rotations from the original vector (same axis of rotation).

up to 30 minutes of exposure time. This makes our setup perfect to do real-time processing in observational astrophysical applications.

Verdict and Next Steps

Those results in the validation loss (-0.85) means that we have a mean angular error of about $\sim 30^{o}$. A first analysis to the different galaxies showed that as we can see in the left image of Figure 3 the disc galaxies have an almost perfect prediction, and the rounded and irregular galaxies have the worsts predictions due to having less defined angular momentum, and this is what made our mean error higher than if we just include disc galaxies.

During the project, we also noticed that the data is one of our limiting factors. We need to have more data, restrict the data to have just the adequate morphological types of galaxies, and normalise differently to have a sense of how strong the angular momentum is and not just where it is pointing at.

Acknowledgements

This project has been a real inspiration for me to continue exploring the machine learning world and solve real-world problems with the tools that I learned here. I want to thank the SoHPC to bring me the opportunity to develop this project and give special thanks to my mentor Georg and to Marc Huertas, Arianna di Cintio and Christopher Bryan Brook at the institute of astrophysics in the canary islands to help me on this journey. Without them, any of this would have been possible.

Data and source code

If you liked the project and want to try it yourself, all the data and sources can be found at my GitHub page. The link is in the More info section (andreuva/DNN).

References

[1] Liang Wang et. al. "NIHAO project I: Reproducing the inefficiency of galaxy formation across cosmic time with a large sample of cosmological hydrodynamical simulations." Monthly Notices of the Royal Astronomical Society, November 2016.

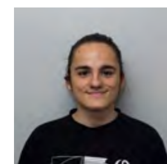
[2] Kaiming He et. al. "Deep Residual Learning for Image Recognition" Microsoft Research

[3] Alex Krizhevsky et. al. "ImageNet Classification with Deep Convolutional Neural Networks".

PRACE SoHPC Project Title
Object Detection Using Deep Neural Networks – AI from HPC to the Edge

PRACE SoHPC Site
IT4Innovations National Supercomputing Center, Czech Republic

PRACE SoHPC Authors
Andrés vicente, Institute of astrophysics of the canary islands, Spain.



PRACE SoHPC Mentor
Georg Zitzlsberger, IT4Innovations, Czech Republic.

PRACE SoHPC Contact
Andres, Vicente Arevalo, IAC
Phone: +34 675 139 063
E-mail:
andres.vicente.arevalo@gmail.com

PRACE SoHPC Software applied
Tensorflow, Jupiterlab, Pynbody.

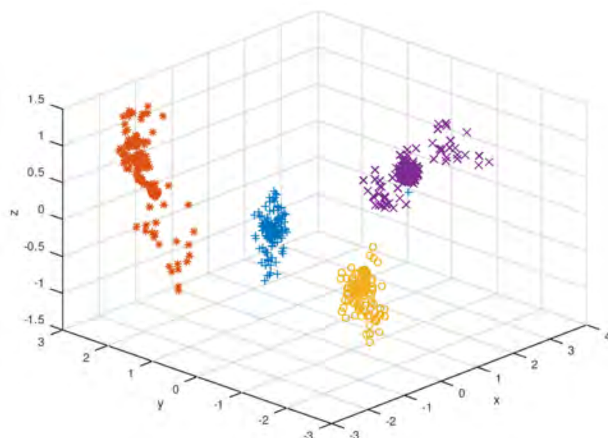
PRACE SoHPC More Information
pynbody.github.io
https://github.com/andreuva/DNN

PRACE SoHPC Project ID
2009

A visualizing tool for the data outputs of various molecular simulation techniques.

Visualization of Molecules

Denizhan Tutar



Visualization of molecules and atomic clusters is an important step in many scientific applications. This project aimed for the utilization of OpenGL rendering pipeline and thread based parallelization in Python 3 for lightweight visualization of molecules.

Proper and light-weight visualization of molecules, atom clusters or orbitals is an indispensable part of a bunch of research fields such as chemistry or materials science. Although there are successful tools available, the related scientific community has diverse needs that cannot be addressed altogether by one single tool and development of novel tools is a vivid and interesting domain for many researcher.

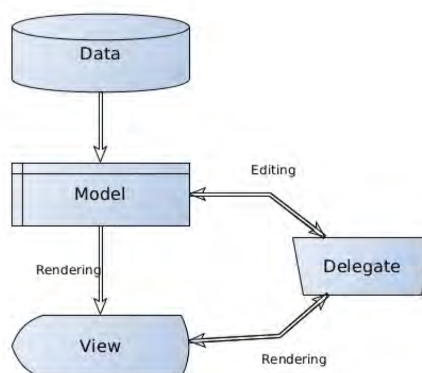
Motivation of the project

The motivation of the project was to develop a molecular visualization tool that is lightweight and that can be used in many popular platforms. The end product was also intended to be easily used from the terminal, and provide a practical export of data for some other tools such as Gnuplot.

Overall structure of the program

The only programming language used in the project is Python 3. Python 3 is widely used in scientific community, and rich in term of high level libraries. Our application includes such libraries: Numpy for fast numerical manipulation, PyOpenGL as a binding to OpenGL application program interface (API) that handles rendering, Pyglet for user interactions, and Glumpy to bind them all.

Visualization from a data file has three steps: reading data to memory (I/O), computing list of vertices from data, and rendering. These tasks can be performed in many different ways that each may have their own advantages and disadvantages, that is why there is not a single common visualization pipeline template. But there are some popular approaches that we also implemented.



A notable example of such an approach is Model-View-Controller architecture that divides the application in three interconnected parts. Here, Model stands for the data in the memory, View is for visual output on the screen or to be saved, and Controller (or delegate) performs editing following the commands of the user. We followed a similar approach but used numpy arrays as model, Glumpy to provide View in general and handlers in Pyglet's window module as controller for its simpler user interaction capabilities.

A main feature of our program is to provide user interaction to manipulate the image. This involves zooming in and out, rotating the image, and changing atomic radii.

The Rendering Pipeline

The rendering is the essence of visualization tasks. OpenGL is a widely used API, mainly written in C++ and C, and many languages provide wrappers to it including Python. Its rendering pipeline starts with a list of vertices, which is used to generate visualisation primitives. Those are later used to compute fragments and pixels, through well structured steps that provides user many ways to interact with the process.

Parallelization Strategy

The first step was to decide which parallelization technique(s) will be used. There are two main approaches we investigated: multiprocessing and multithreading.

In multiprocessing approach, many process are started to run in parallel.

```
Every 2.0s: nvidia-smi                      krtecek: Mon Aug 24 15:38:45 2020

Mon Aug 24 15:38:45 2020
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 440.100      Driver Version: 440.100      CUDA Version: 10.2     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  GeForce MX250       Off  | 00000000:01:00.0 Off |                  N/A |
| N/A   65C    P0    N/A /  N/A |    257MiB /  2002MiB |      4%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|    0      1040      G   /usr/lib/xorg/Xorg                           255MiB |
|    0      1673      G   xfwm4                                          1MiB |
+-----------------------------------------------------------------------------+
```

```
Every 2.0s: nvidia-smi                      krtecek: Mon Aug 24 15:36:30 2020

Mon Aug 24 15:36:30 2020
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 440.100      Driver Version: 440.100      CUDA Version: 10.2     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  GeForce MX250       Off  | 00000000:01:00.0 Off |                  N/A |
| N/A   68C    P0    N/A /  N/A |    267MiB /  2002MiB |     42%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|    0      1040      G   /usr/lib/xorg/Xorg                           261MiB |
|    0      1673      G   xfwm4                                          1MiB |
|    0     24497      G   python3                                        3MiB |
```

Screenshots from nvidia-smi tool. Left image is before our application starts, right one is during the rendering.

The Operating System (OS) treats each process roughly like an independent program and assign separate resources. They often communicate via Message Passing Interface or an alternative tool. This approach is more suitable for massively parallel computations that spans many nodes. We did not use this approach because our task is usually not that massive and expected to be done within a single node.

In multithreading approach, two or more threads run in parallel. One of the most important differences of a thread from a process is that they share some of the computational resources, such as memory. This gives rise to an important class of errors, namely race conditions. Race conditions are infamous for being sneaky errors, and many mechanisms are developed to avoid them. One of the such mechanisms, locks, are implemented in Python; but unlike many other languages, most popular python interpreters implemented Global Interpreter Lock (GIL) that restrains all the shared memory from being accessed by more than one thread at a time. This reduces the number of active threads at any given time to one and makes threading an ineffective approach for CPU-bound applications. Still, threading in Python can be quite useful since it can be used to overlap data migration and computation, at least for I/O-bound applications.

Even when only one thread can access data, the OS can switch among threads at any time, and we should enforce threads to wait for each other in some critical sections of the algorithm. We used a bounded semaphore, a counter that is incremented or decremented when a thread reaches a certain section, and made others wait until this thread is done to avoid unexpected results.

The end product

We achieved some of the main targets. The project has a working visualization pipeline, thread parallelism for overlapping I/O with computation by reading data in chuncks and doing prerendering computations in parallel, can transfer the rendering workload to GPU when possible, and contains the most basic functionalities such as zoom and rotation. You can see an example view made with our program.

We used nvidia-smi, a handy GPU monitoring tool, to investigate our GPU usage and be sure that we achieved to transfer the rendering workload into GPU. You can see the GPU-utilisation and Python 3 process in the queue from the following screenshots. The first one is before and second one is during the use of our program. This is reached by according usage of Glumpy and PyOpenGL without extra steps, as those libraries are written to use GPU when available.

Lastly, our tool is capable of rendering multiple scenes in the same file, resulting in an animation. It is also multi-platform.

Discussion & Conclusion

We had reached some of the goals set in the beginning and provided a suitable baseline for further improvements. But there is still some work to do to release a fully functional visualization tool. We can mention some major future works as:

- Increasing the GUI functionality
- Easier use from the terminal
- Visualization of electronic densities as isosurfaces
- Implementing a more advanced parallel I/O capability

References

[1] https://realpython.com/python-gil/ last accessed 30.08.2020

[2] https://glumpy.readthedocs.io/en/latest/ last accessed 30.08.2020

[3] https://pyglet.readthedocs.io/en/latest/ last accessed 30.08.2020

[4] https://numpy.org/ last accessed 30.08.2020

[5] http://pyopengl.sourceforge.net/documentation/index.html last accessed 30.08.2020

[6] https://www.opengl.org/documentation/ last accessed 30.08.2020

Denizhan Tutar

PRACE SoHPCProject Title

Development of visualization tool for data from molecular simulations

PRACE SoHPCSite

IT4I, Czechia

PRACE SoHPCAuthors

Denizhan Tutar, [ITU] Turkey

PRACE SoHPCMentor

Martin Beseda, IT4I, Czechia

PRACE SoHPCCo-mentor

Rajko Ćosić, IT4I, Czechia

PRACE SoHPCContact

Denizhan, Tutar, ITU
Phone: +90 537 573 4632
E-mail: tutard@itu.edu.tr

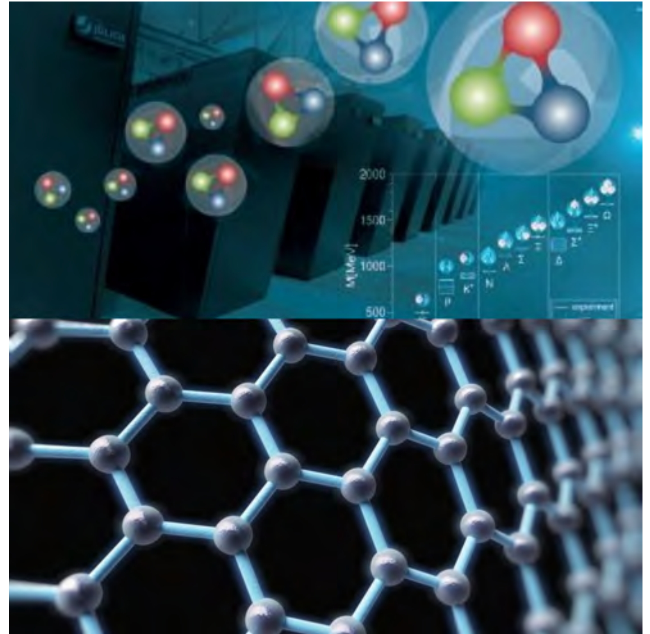
PRACE SoHPCAcknowledgement

Special thanks to our mentors that were very helpful and provided the opportunity to work really close, made possible for the project to met some of the goals.

PRACE SoHPCProject ID

2010

Simulations of classical or quantum field theories take up a large fraction of the available supercomputing resources worldwide



High Performance Quantum Fields

Aitor López
Anssi Manninen

The efficient computation of quantum field theories demand the correct usage of HPC architectures and subtly chosen methods for tasks such as matrix inversion.

Lattice discretized quantum chromodynamics

The path integral formalism offers a powerful tool to evaluate physical quantities from quantum mechanics and quantum field theories. In the case of quantum chromodynamics (QCD), the path integral evaluates all possible configurations of continuous scalar fields over time interval T.

To make the computation of path integral computationally feasible, the lattice quantum chromodynamics (LQCD) is introduced. The LQCD consists of from 4-dimensional lattice that allows spatial and time coordinates to take only discrete values in the lattice sites. In the lattice, each site represents quarks fields and links connecting sites gluon (gauge) fields. Such a set-up enables one to calculate the path integral via machinery. The quantities inferred from the LQCD simulation play a significant role when investigating the correctness of the prevailing theory.

In this project, hadron masses are extracted from path integral data acquired using different field configurations. Consequently, the determined mass values are utilised in the fitting analysis to tune the bare masses of strange and charm quarks. The tuned bare mass values work as preconditioners which transform the output quantities to correspond experimentally measured physical values.

Introduction

The path integral formalism in field quantum field theories allows one to acquire the output of correlators (Greens functions) computationally. To make the calculations of integral over continuous scalar fields feasible the space geometry is discretised with the LQCD model. As a result, the integral can be approximated by generating a set of sample field configurations and averaging the outputs. The configurations are generated with Monte Carlo methods using a statistical weight of e^{-S}, where S is the action of the configuration.

To ensure the correct physical behaviour of the LQCD simulations, the outputs should always correspond as close as possible to real physical quantities. Two factors affecting the out quantities are lattice spacing a and quark bare mass values. One way to tune the input values such as strange quark is to analyse the quantities from meson correlators.

Methods

To get the meson correlators the quark and anti-quark propagators are contracted with so-called 2-points functions (separately for each time step). The contraction function can have an additional part which includes prior information (smeared contraction) about the relative spatial distribution of the quarks in the investigated hadrons. The hadrons corresponding different quantum numbers are created by combining the gamma matrices γ_i with the measured field operators $\mathcal{O}$. The used mesons in this work were pseudoscalar and vector mesons.

For the mesons correlators C the expected theoretical output is of form

$$\langle C(t)\rangle = \sum_i a_i e^{-tm_i},$$

where a_i is constant and m_i mass of nth state (in a unitless form). In consequence of used periodic boundary condi-

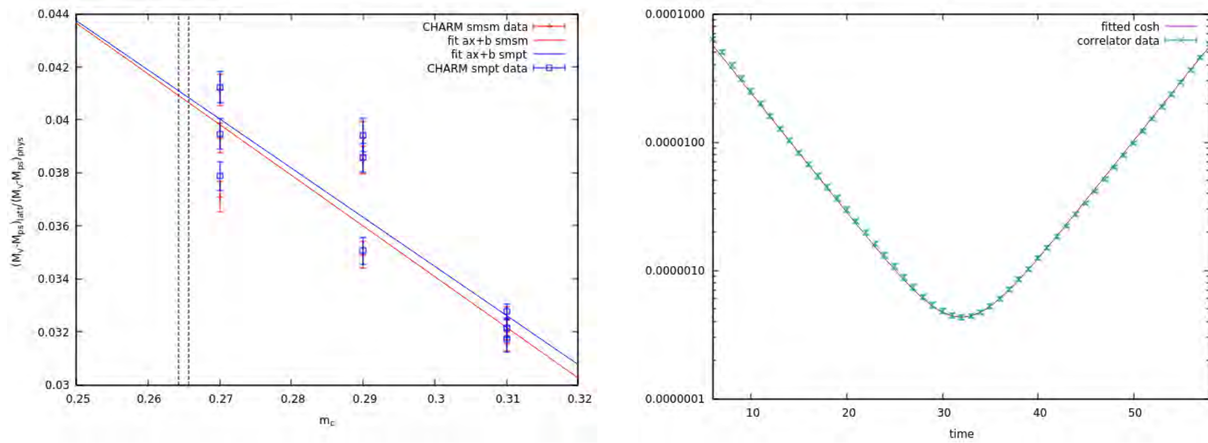


Figure 1: Hyperbolic cosh fit to meson correlator data (rigth) and Charm quark bare mass vs. the relative splitting term calculated with masses determined from meson correlators (left).

tions of LQCD the actual form of the correlator output is cosh function over lattices interval $t = 1 \ldots 64$. After averaging the correlator data, the ground state mass can be carefully extracted from the exponential fit due to its slow exponential decay. The fits were done with Gnuplot's weighted nonlinear least-squares Marquardt-Levenberg algorithm by using standard errors of average values as weights.

An important thing to notify while calculating the weights was the correlation between the configurations. Therefore straight calculation of standard error would not describe the weights of the fits properly. A simple way to reduce the amount of correlation is to use blocking, i.e. bin each N number of most configurations to form one datapoint. The second method to take into account the correlations is to calculate the standard errors by jackknife analysis.

Since the determined masses are dimensionless quantities depending on lattice spacing a; the tuning is done by comparing the relative charm meson splitting

$$\frac{M_V - M_{PS}}{M_V},$$

where M_{PS} is pseudoscalar meson mass and M_v is vector meson mass. The relative splitting term is approximately behaving linearly w.r.t. bare mass, hence the bare mass value producing physically valid relative splitting term can be extrapolated from linear $(a + bx)$ fit in bare mass in the function of relative splitting.

Knowing the meson splitting term $M_V - M_{PS}$ being also linearly dependent on charm bare mass value and using the relation $M_{phys} = aM_{latt}$ allows the extrapolation of the lattice spacing a for the previously tuned charm quark bare mass.

Results

The analysis was made with data produced from HPC LQCD simulations in advance. The charm meson correlator dataset consisted of nine different ensembles configured with different bare mass values. For each correlator ensemble, the charm pseudoscalar and vector masses were determined. With the obtained mass values the bare mass value yielding physical relative splitting was established (fig. 1).

Once the lattice spacing was revealed, further analysis could be made with strange quark masses. The strange bare mass corresponding the target energy was extrapolated from linear fit applied to the determined strange quark vector masses.

Conclusion

The LQCD offers a salient tool to investigate the behaviour of QCD physics. The quality of the state of art LQCD simulations highly depends on the used HPC architecture. The parallelization allows LQCD simulation parameters such as lattice spacing take reasonable values to scale the errors of LQCD simulations relatively small.

Due to historically increasing efficiency of the HPC architectures, the LQCD will offer irresistible opportunity to study the physics of quarks and gluons even more versatile in the future.

Simulation of Quantum Monte Carlo for Carbon Nanotubes

Graphene is a two-dimensional material whose thickness is one atom and arranged on a hexagonal ("honeycomb") lattice, as you can see in the figure 2 . The study of this material is increasing due to its unusual properties. These range from extreme mechanical strength and lightness, through unique electronic properties to several anomalous features in quantum effects.

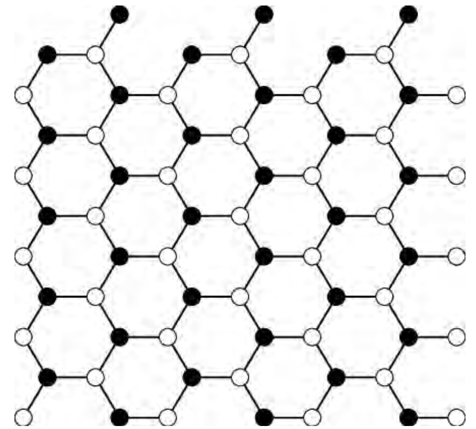


Figure 2: hexagonal Honeycomb lattice. Figure content uploaded by Yasumasa Hasegawa

Experimental research on graphene includes simulations where the properties of strongly interacting matter are studied and can be based on the experience gathered in the Lattice QCD. These simulations require, for example, the repeated computation of solutions of extremely sparse linear systems and update their degrees of freedom using symplectic integrators.

At the Jülich Supercomputing Centre (JSC) in Germany, they have devel-

oped Hybrid Monte Carlo (HMC) methods to calculate graphene's electronics properties using lattice-discretized quantum field theory, a model widely used to make particle physic predictions using HPC.

The objective of this summer project was to understand how the algorithm works, the parts that surround it and which parts of the code present more complexity to the processor. Specifically, we have modified the algorithm in charge of solving the linear system involved so that it runs in parallel on the GPU.

Introduction

In order to investigate the basic properties of the Hubbard model in this geometry, Hybrid Monte Carlo (HMC) simulations are used.[2] The underlying Hamiltonian, after Hubbard-Stratonovich transformation and introduction of pseudofermions, is

$$\mathcal{H} = \frac{1}{2}\delta\phi^T V^{-1}\phi + \chi^\dagger (MM^\dagger)^{-1}\chi + \frac{1}{2}\pi^T\pi \quad (1)$$

where π is the real momentum field, ϕ is the real Hubbard field, χ is a complex pseudofermionic vector field, δ is the step size in Euclidean time dimension weighted by the inverse temperature the system, V is the interaction potential and M is the fermion operator.

The basic HMC algorithm now generates π and an auxiliary complex field ρ according to a Gaussian distribution $e^{-\pi^2/2}$, respectively $e^{-\rho^\dagger\rho}$. Then the pseudofermionic field is obtained as $\chi = M\rho$. With these starting parameters and an initial field ϕ a molecular dynamics trajectory is calculated and the result is accepted with the probability $\min(1, e^{-\Delta\mathcal{H}})$. $\Delta\mathcal{H}$ the difference in energy resulting from the molecular dynamics.

Solver

Note that equation (1) involves a matrix "inversion" via the matrix equation

$$(MM^\dagger)x = b \quad (2)$$

where $MM^\dagger$ is a large matrix, b is a known vector defined by the states we are considering ($b = \chi$), and x is an unknown vector. This equation must be solved very often, making it beneficial to optimize the run time of the linear solver.

Initially at JSC, they used two iterative schemes to estimate the solutions x of the above equation, using a Conjugate Gradient-based solver (CG) for a multi-CPU architecture using OpenMP (Note that the $MM^\dagger$ matrix is Hermitian and positive definite) implemented in C++.[3] The first, single precision CG is used to solve well-conditioned linear systems and it is good enough to obtain a precision of 1e-4 to 1e-5, but as it is a non stationary solver and the precision needed varies with every iteration, when you need more precision the standard CG is replaced by a flexible Generalized Minimal Residual (FGMRES) with a single precision solver as preconditioner (like CG). Due to the large number of operations that have to be performed (mostly due to the size of $MM^\dagger$) both solvers present a bottleneck in the execution. For this reason, we have adapted both solver codes to run on the GPU (with OpenACC) and thus reduce the execution time and improve the performance of the HMC algorithm.

Results

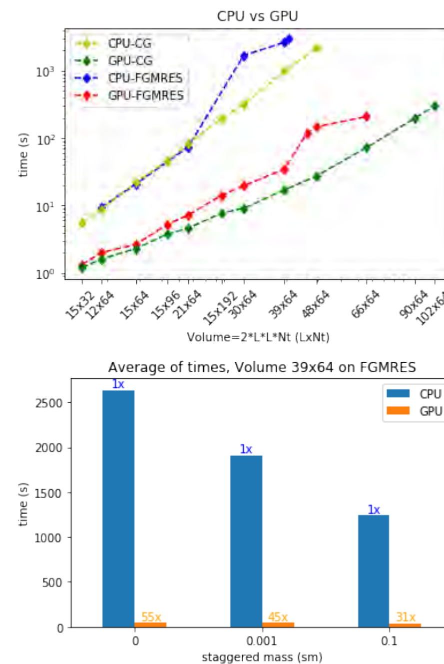


Figure 3: Comparison of times of executions on CPU and GPU. Free volume (up). Fixed volume (down).

The results presented here have been run on single nodes 2×Intel(R) Xeon(R) CPU E5-2680, 40MB Cache per node for experiments on the CPU and the GPU compute nodes feature four NVIDIA V100 SXM2 GPUs. The parameters associated with the interaction potential V, of the Hamiltonian of the equation (1), have been set to $\beta_\mathcal{K} = 2.5$ for temperature and $U_\mathcal{K} = 8$ for Hubbard ratio.

To show the improvement in performance that we obtain in both solvers when we use the GPU in their execution, we've run two experiments for each solver. In the first one, we move the hexagonal lattice volume of the graphene, $L \times L$ with N_t timesteps. In the Figure 3 (up) we can see the execution time required for the solvers CG_single and FGMRES, respectively. In a second experiment, we leave the volume fixed and change the staggered mass (ms), another parameter that intervenes in the creation of the matrix M, in the Figure 3 (down) we can evaluate the Speedup that we get, reaching improvements of up to 55x in FGMRES solver.

Conclusion

It should be noted that we have worked with a research team that is at the forefront of quantum field simulations and that uses a scientific code that without HPC technology it would be impossible to obtain a complete simulation.

References

[1] C. Gattringer and C.B. Lang (2010). Quantum Chromodynamics on the Lattice.

[2] S. Krieg, T. Luu, J. Ostmeyer, P. Papaphilippou, and C. Urbach (2019). Accelerating Hybrid Monte Carlo simulations of the Hubbard model on the hexagonal lattice.

[3] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, H. V. der Vorst, (1993) Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods.

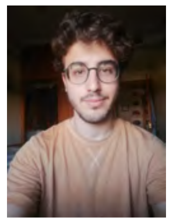
PRACE SoHPCProject Title
High Performance Quantum Fields

PRACE SoHPCSite
JSC (Jülich Supercomputing Centre), Germany

PRACE SoHPCAuthors
Aitor López, Spain
Anssi Manninen, Finland

PRACE SoHPCMentor
Stefan Krieg, JSC, Germany
Eric Gregory, JSC, Germany

PRACE SoHPCContact
Aitor, López Sánchez
Phone: +34 697 56 17 64
E-mail: aitor.lopez1@um.es
-
Anssi, Manninen
Phone: +358 505324356
E-mail: ansman@uef.fi


Aitor López

Anssi Manninen

PRACE SoHPCAcknowledgement
Thanks to our mentors and the entire SoHPC team for this fantastic experience.

PRACE SoHPCProject ID
2011

FMM-GPU Melting Pot

Igor Abramov and Josip Bobinac

Graphics processing units (GPUs) are not used for video games only! They have their applications in science for intensive computational tasks. Simulation of a system of a large number of particles is one such task. It can be simulated using an algorithm called the Fast Multipole Method (FMM). However, to make it really fast on the GPU, the data layout needs to be adapted.

Simulation of the dynamic system of particles or the N-body problem has interested scientists for the last several hundred years. The main task at every time step is moving each particle from current position according to its velocity, followed by updating its velocity according to the force exerted by other particles. Due to the lack of analytical solution for systems containing more than three objects, a lot of numerical methods were developed. Imagine computing the forces on each planet of our solar system caused by all the other planets. For each of the planets, one would need to add up the contributions to the total force of all the other planets, one at the time. Assuming all the data is available, the task is computationally trivial. However, in typical simulations, where the particles are counted in trillions, computing anything with this approach would take another several hundred years even on the most sophisticated computer architectures. To avoid such scenarios, there is a need for more sophisticated methods. The Fast Multipole Method (FMM) emerged as one of the most interesting because of it's level of complexity and accuracy.

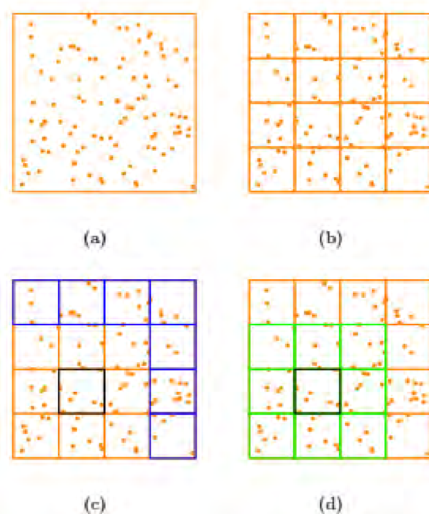


Figure 1: (a) simplified 2D view of the domain. (b) example of the domain division. (c) target box in black and well-separated boxes in blue. (d) neighbouring boxes in green.

The method is based on the idea of approximating groups of particles, based on their locations, into pseudo particles that represent their groups to simplify the computation process.You are probably wondering how big of an approximation error these representative pseudo particles create. The greatest benefit of the FMM is that by doing this approximation, accuracy does not drop off significantly. In fact, for a certain amount of particles in the system, FMM might even lead to a smaller error due to its reduced numerical error i.e. numerical error is naturally present in every simulation due to hardware limitations and when added to its approximation error the sum is smaller than the numerical error of the naive approach.

How does FMM work?

The mentioned groups of particles are split into two categories - neighbouring and well-separated ones. Well-separated groups are represented by a single expression called the multipole

expansion and the neighbouring by a local expansion. Multipole expansion is an approximation of the impact the considered group has on its environment. When the force is being computed at a certain point, instead of accounting for each particle in the group, only this single expression is be considered. By now you probably connected these expressions and the pseudo particles mentioned in the previous paragraph. They are equivalent terms in this scope. Overall, the FMM algorithm can then be implemented by splitting it into several operators. For more details, we refer the reader to Albert's article.[1] The slowest, and therefore the operator that calls the most for optimisation, is the so called Multipole-to-Local operator which computes the contributions of all these group representative expressions onto the expression of interest. To do so in the mathematically least complex manner, alignment by rotation and shift to the target expression need to be done. This way, the algorithm is at the mathematical limit of the complexity. However, there are numerous ways to implement the same algorithm. The final performance depends on how the data is stored and accessed. The goal was to implement the rotation in a way that the benefits of parallelization for the GPU are maximised.

Benefits of GPU parallelization?

To understand how to reap these benefits, one first needs to understand the fundamental difference between a standard computer or the CPU and the GPU. CPUs typically consist of a few flexible and powerful cores, designed for handling a wide variety of tasks in a sequential manner. On the other hand, a typical GPU's high count of comparably weak and simple cores makes them very powerful in performing highly parallel tasks. Practically, this means CPU outperforms the GPU in tasks like listening to audio material, browsing the web and all other common functions of a computer. However, the highly parallelizable tasks are where the GPUs exhibit the best performance e.g 3D graphics processing. To be highly parallelizable, it must be possible to break the task down into numerous simple, data independent tasks that can than be sent to the simple cores of the GPU. As stated in the beginning, GPU's are also used in science. To reap the benefits when doing the rotation in the FMM, the data used needs to be stored and accessed in a highly parallelizable manner.

To parallelize or not to parallelize?

Recall the multipole expansions. These are approximations to the specific order represented in memory by

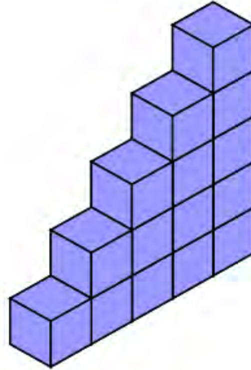


Figure 2: Multipole expansion representation in memory. Each block holds a (complex) coefficient.

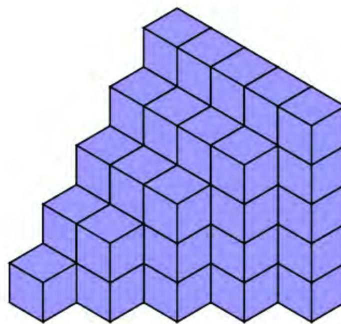


Figure 3: A set of rotation matrices forming a pyramid structure.

The size of this set and the approximation accuracy directly depend on the approximation order. Each higher order term requires storing one more coefficient than its lower order predecessor. Hence, the data structure used to contain such an expansion can be visualised as a triangle of blocks, where each block holds a coefficient.

Next, to rotate the expansion, rotation matrices are used. These matrices stem from Wigner rotation matrices and further clarification is omitted due to the focus on parallelization. Important fact to note is that these are square matrices whose dimensions get incremented with the order.

Therefore, the containing data structure is a pyramid as shown in the Figure 3 below.

The rotated multipole expansion is obtained by multiplying the two structures as shown in Figure 4. Since typical simulations are three-dimensional, there is 189 multipoles that need to be rotated. Some of them share the same angle for rotation so there is twenty-six angles to be used. Practically, for the processor, this means doing multiplications with 26 different pyramid structures of 189 different triangles. This is not a kind of task a GPU would like. GPU would like if it had to rotate all the triangles with the same pyramid. However, it can be shown the same multipole rotation effect can be achieved by taking several smaller rotation steps. The first of these smaller steps is rotating each multipole by a fixed angle of 90 degrees. We put our attention to this step. Rotating all the multipoles by the same angle means using only a single pyramid which is now a highly parallelizable, GPU-friendly task.

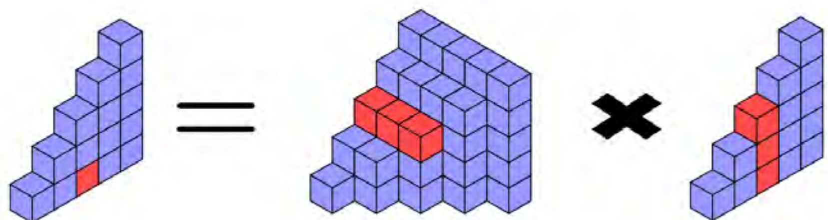


Figure 4: Obtaining one coefficient of the rotated multipole. Each of the three pyramid entries multiply the triangle entries and the results are summed and stored as shown.

How to parallelize?

When preparing the code for the GPU, it is crucial to keep in mind its basic working principles. The cores are a hardware concept. From a software perspective, GPUs can launch a high number of threads, where each thread belongs to a core. This high number of threads is distributed into blocks of threads due to their corresponding cores being physically grouped together. The threads inside a single block are dependent on their block colleagues. Only when all the threads are done with their work can the new work be assigned to any of them. One unit of work that this block does is called the "lock-step". Since the threads inside a block typically have the same power, if we were to assign different workloads to members of the block, the fastest threads would spend some time idle. Therefore, to make the best out of available computational resources, balancing the workload for the members of the block is essential. Furthermore, once the processor reads some data from the main memory, it is stored in the fast memory local to the processor. Ideally, all the data would be stored in the registers because it would be able to access it much faster than from the main memory. However, this is impossible since their capacity is limited. If data stored in the register does not get used for a short time and there are other processes happening, this data will get flushed out to make space for newly needed data. Due to this difference in access time from different parts of memory, it reusing the data loaded into registers before it gets flushed out brings significant performance benefits.

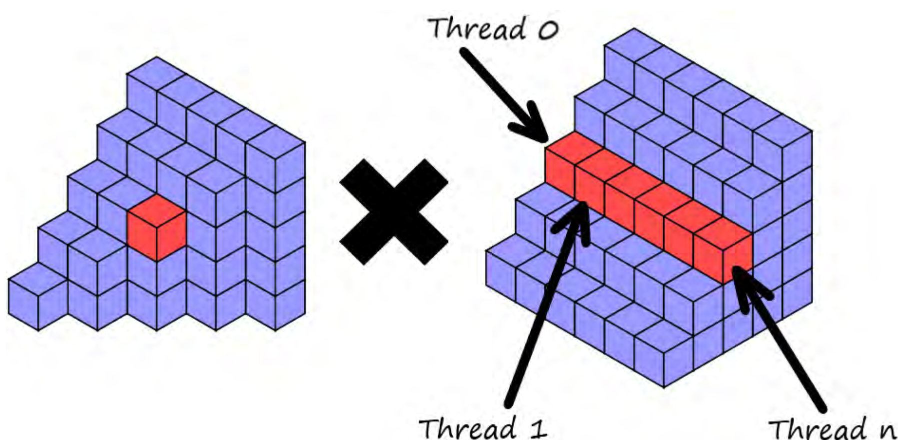


Figure 5: Obtaining one coefficient of the rotated multipole. Each of the three pyramid entries multiply the triangle entries and the results are summed and stored as shown.

So what does that mean in our case?

Multiplication requires several loads of each multipole coefficient. Once it is loaded from the main memory, multipole coefficient is only used for a single operation, making it hard to reuse. However, the multiplication can be set up in a way that each pyramid elements is only accessed once. This allows the machine to reuse the element loaded in the registers, avoiding repeated loading from the slower, main memory. If we were to rotate the multipoles one at a time, each pyramid element would be accessed once for each multipole. This is not an optimal scenario. To minimise the pyramid access frequency, multipole triangles are stacked together. Once a particular element of the pyramid is accessed, each thread in the block gets a copy and uses it to multiply the corresponding element of the multipole triangle. This way, once the pyramid element is loaded into the registers, it is reused as described above. This stacked multiplication occurs in the same fashion as for the single multipole triangle. Each thread works on their own triangle in a stack and stores the result in the same fashion as in Figure 4. Ideally, we would put stack all the triangles together and rotate them at once. However since GPU architecture has its practical limitations, there is a sweet spot when it comes to stacking the triangles. This can be found by running a series of tests with a varying size of the stacked structure.

Future Work

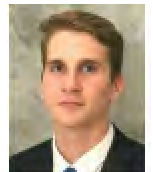
Current method code was optimized for CPU usage and therefore it is using many constructions, that are not available for GPU compilation at the moment (like std smart pointers, memory allocating primitives, etc.). Process of rewriting method code to make it possible to run it on GPU took much more time than it was expected. That's why in the future, the debugging of the code on the GPU should be finished to find the optimal size of the stack of triangles. This idea of reusing data that can be reused can be applied to other operators in the FMM to boost performance.

References

[1] Albert Garcia (2015). Parallel FMM on a GPU a CUDA/C++ love story - Summer of HPC project reports 2015.

PRACE SoHPCProject Title

Got your ducks in a row? GPU performance will show!



Igor Abramov

PRACE SoHPCSite

Jülich Supercomputing Centre,Germany

PRACE SoHPCAuthors

Igor Abramov, Swiss Federal Institute of Technology in Lausanne (EPFL), Switzerland
Josip Bobinac, TU Wien, Austria

Josip Bobinac

PRACE SoHPCMentor

Dr. Ivo Kabadshow, JSC, Germany

PRACE SoHPCContact

Ivo Kabadshow, JSC
Phone: +49 2461 61-8714
E-mail: i.kabadshow@fz-juelich.de

PRACE SoHPCSoftware applied

C++, CUDA, GCC

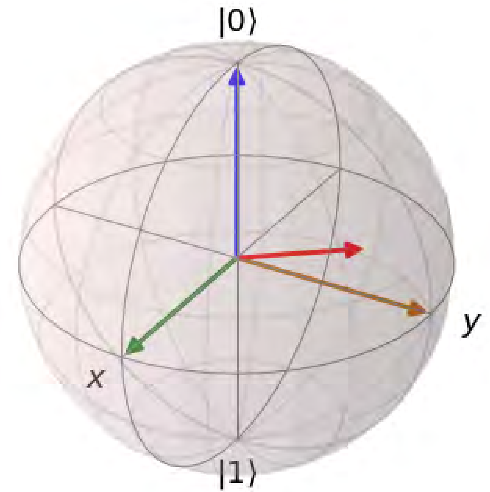
PRACE SoHPCAcknowledgement

We would like to express our gratitude to our mentor, Ivo Kabadshow, who was repeating method descriptions and speeches about optimisations until we were not fully sure that we understand everything and then were calling him next day with twice more questions. Thanks Ivo for maintaining online working format with the same attitude and support as if we presented in Jülich in person. Also thanks to PRACE and the SoHPC organizers for their effort which made it possible to do summer program this year regardless of all the complicating circumstances.

PRACE SoHPCProject ID

2012

Quantum Computing



Benedict Braunsfeld, Sara Duarri Redondo

An implementation of a genome pattern matching application focused on DNA sequences running on a quantum simulator.

Pattern matching is in computer science the act of checking a given sequence of tokens for the presence of the constituents of some pattern. Finding these patterns can help in all kinds of tasks i.e. behavioural patterns, genome patterns or connecting patterns we can not grasp with our eyes. In the case of DNA an genetic patterns, these approaches, usually run on super computing clusters, take days to finish due the amount of data, and usually with an approximate solution. Due to the high volume of data that is parsed to the sequence alignment algorithms they often use heuristics methods. This permissiveness with the solution and the high volume of information makes these problems attractive to solve in a quantum computer, because of the high capability to store data with quantum bits (qubits).[1]

When comparing to classical bits, qubits can store a 1 or 0 but also be both in state $\alpha|0>$ and in state $\beta|1>$ at the same time before measurement. This property (superposition) allows qubits to store more information per qubits (compared to classical bits), due to the memory growing up exponentially as the number of qubits augments instead of linear.

In our project we mainly focused on the string encoding and comparison. For the alignment we decided to work with a naive approach, but of course further improvements could be done using a better algorithm. It is important to remark that because of the lack of availability of real quantum computer these analyses were carried on a simulator from IBM. We chose to stick with the qiskit package from IBM instead of the Rigetti forest API.[1–4]

1 Encoding

To use quantum computing to compare multiple genetic strings, the strings need to be represented by quantum states first. To encode a string into a quantum state a superposition and register qubits are used. The amount of qubits used for the superposition depends on the size of the encoded string. The amount of register qubits depends on the amount of different characters in the string. The register qubits stores information of the position of the element in the string and information about the element itself. To encode a random string the information of the position and type of every character has to be translated into binary. For positions, the number referring to the position, starting counting at 0, will be converted to binary. For the values, each value would be assigned to a number, starting from 0 and increasing from 1 to 1. Those numbers would also be converted to binary.
For example to encode the string '-M-M' four positions need to be represented and 2 of them had to differ from the other two. Therefore a superposition of two qubits which creates four states and one register qubit is required.

Figure 1: Quantum state representation of the string '-M-M'.

To put a qubit in superposition a Hadamard gate is applied (equation 1. The Hadamard gate or H-gate maps the basis states $|0>$ and $|1>$ to $\frac{|0>+|1>}{\sqrt{2}}$ and $\frac{|0>-|1>}{\sqrt{2}}$. This means that a after being measured it has equal probabilities to become 0 or 1.[5]

$$A = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \tag{1}$$

To control the register qubit depending on the superposition a Toffoli or CCNOT gate is applied. The CCNOT gate is a 3-qubit gate which flips the third qubit if the first two qubits are both 1.[6] To control and flip a single qubit a Pauli-X gate can be used (equation 2)

$$A = \frac{1}{\sqrt{2}} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \tag{2}$$

To apply this on the genome problem and a multi-qubit problem the qubits are initialised in a quantum circuit. After initialising the circuit the qubits

used to represent the position of every character in the target string are initialised as a state of the superposition. The number of states in a superposition scales exponentially to the the amount of qubits. The register qubits are all 0s. Afterwards the register qubits need to be flipped at certain positions (in the example at positions with M). Therefore we apply Pauli-X gates on the state in the superposition that is representing the position in the string to create a state of all 1s. Afterwards a NCX gate (same as CCNOT but for infinite number of qubits and not only two) is applied to flip the register qubit. This is done for every position in the target string that needs to be flipped.

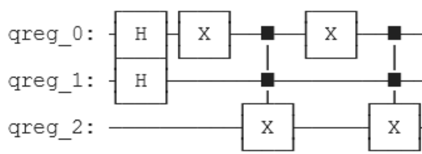


Figure 2: Quantum circuit representing the encoding of the string '-M-M'.

2 String comparison

The string comparison is based on a simple idea. All quantum circuits are initialised at 0 for every Quantum Register. Once operations are applied to encode the string, the quantum circuit is changed in a certain order. Since all quantum computations have to be reversible, a reversed quantum circuit of a second string could be applied to the first quantum circuit. If both strings are the same the combined quantum circuit should end up being 0. The more different the strings are, the more the combined quantum circuit differentiate from a circuit with all 0s. For example, if '—-' and 'MMMM' are compared, the resulting registers will all be 1 when measured.

3 Application

To apply this, an user-friendly application was developed, mainly to do pattern matching with DNA. The application needs two arguments arguments: –genome (file where genome is stored) –reads (file with DNA reads that should be match to the genome). Further options are: –threads (allows using multiple threads, default is 1), –code (in case it is not a DNA sequence other code than ATGC can be specified), –rc (for DNA, check also for reverse complement strings).

4 Performance

In figure 3 the results for a strong scaling and weak scaling experiment are shown. For the strong scaling experiment a problem size of 2048 bits and 10 reads of the same size were tested with a different numbers of CPUs. Since the difference between the improvement in time is not as big for 16 to 32 as for 2 to 4 CPUs the limit of how much more CPUs will increase the performance is about to be reached. At this point the shared workload is reduced that much that not all processors are fully occupied. For the weak scaling both number of processors and the problem size increase (for each problem size the size of the reads is increased equally). In this case it is not scaled up to big enough problems to see, if the memory-bound of the application is reached.

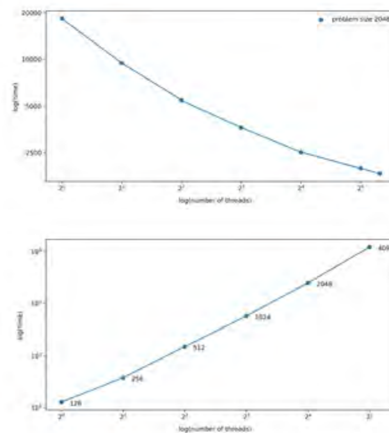


Figure 3: Scaling experiment on one node with 40 threads. Strong scaling on top and weak scaling on the bottom.

5 Results and Discussion

In this project an application that does pattern matching for DNA sequences was developed. The main computation is developed to run on a quantum simulator. This was a first approach that is not fully tested, but nevertheless, is potentially ready-to-use. Something that would improve the application performance wise would be that qiskit allowed for their code to run in different nodes. With this, a better improvement of the application could be done using HPC systems. In addition to this, and more focused on an application for DNA alignment, using an algorithm instead of a naive approach for aligning the reads would reduce the number of comparisons, and therefore improve performance.

References

[1] A. Sarkar, Z. Al-Ars, C. Almudever, and K. Bertels, "An algorithm for dna read alignment on quantum accelerators," 2019.

[2] H. Abraham, AduOffei, R. Agarwal, I. Y. Akhalwaya, G. Aleksandrowicz, and T. Alexande, "Qiskit: An open-source framework for quantum computing," 2019.

[3] R. S. Smith, M. J. Curtis, and W. J. Zeng, "A practical quantum instruction set architecture," 2016.

[4] P. J. Karalekas, N. A. Tezak, E. C. Peterson, C. A. Ryan, M. P. da Silva, and R. S. Smith, "A quantum-classical cloud platform optimized for variational hybrid algorithms," *Quantum Science and Technology*, vol. 5, no. 2, p. 024003, 2020.

[5] A. N. Akansu and R. Poluri, "Walsh-like nonlinear phase orthogonal codes for direct sequence cdma communications," *IEEE Transactions on Signal Processing*, vol. 55, no. 7, pp. 3800–3806, 2007.

[6] D. Aharonov, "A simple proof that toffoli and hadamard are quantum universal," 2003.

PRACE SoHPCProject Title
Quantum Genome Pattern Matching using Rigetti's Forest API

PRACE SoHPCSite
ICHEC, Ireland

PRACE SoHPCAuthors
Benedict Braunsfeld, Sara Duarri Redondo,

PRACE SoHPCMentor
Myles Doyle, ICHEC, Ireland

PRACE SoHPCContact
Leon, Kos, PRACE
Phone: +12 324 4445 5556
E-mail: leon.kos@lecad.fs.uni-lj.si

PRACE SoHPCSoftware applied
Qiskit

PRACE SoHPCMore Information
www.qiskit.org

PRACE SoHPCAcknowledgement
For their support and organisation we are thankful to Myles Doyle, ICHEC and PRACE

PRACE SoHPCProject ID
2013

Matrix exponentiation on GPUs

Pablo Martinez and Theresa Vock

Scientific software runs on supercomputers. However, having a supercomputer is useless if the software is incapable of taking advantage of such power. Our work is focused on developing matrix exponentiation software that takes advantage of huge hardware resources.

C.Frésillon/ IDRIS /CNRS Photothèque

Supercomputers are an extremely powerful and majestic piece of hardware. The computer science world is, however, different from how everyone could imagine it. Despite having a supercomputer, if the software is not properly tuned and optimized, the supercomputer power will be completely wasted, because the software would be unable to use the astonishing power of such an amazing machine. The *deMon2k*[1] is a scientific software that is used for density functional theory (DFT) calculations. Sometimes, *deMon2k* will need to perform a matrix exponentiation. Such operation is very expensive and is very well suited to be optimized for an HPC (High Performance Computing) environment. Our work focuses on developing a matrix exponentiation that properly exploits the hardware resources in a supercomputer to achieve the greatest performance possible. We develop a CPU, a GPU, and a multi GPU version and we do a comprehensive yet deep study of the performance obtained.

Introduction

CPUs are the core of every computer. It performs all the processing to make a computer work. We are very used to it because every computer or phone has a CPU. GPUs are also present in every computer nowadays. But, what differentiates a CPU from a GPU? Let's have a visual reference.

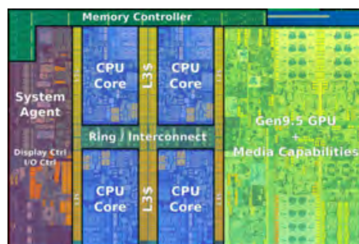


Figure 1: Intel Core 7th Generation die[2]

In figure 1, we clearly see the CPU has 4 cores and an integrated GPU. Each CPU core is pretty big and takes a significant amount of space of the die. Looking at figure 2, we see a different scene. The GPU also has cache memory (we can see the shared L2 in the picture) and has the concept of cores. But a GPU has so many cores that we can barely see them on the figure! In the case of an RTX 2080 Ti, we have 4352 cores. Amazing, isn't it? However, we can't really compare CPU and GPU cores, because they are not and do not behave the same. A CPU core is much more powerful, but a GPU has many more cores than a CPU. If we have many independent operations (such as linear algebra for dense matrices), a GPU usually overcomes the CPU, but in many other scenarios, CPU performs better than a GPU, even though it has less cores.

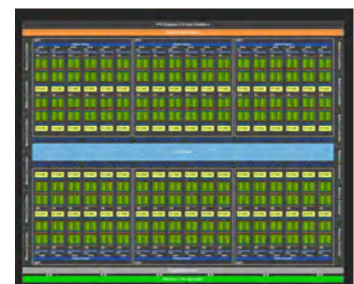
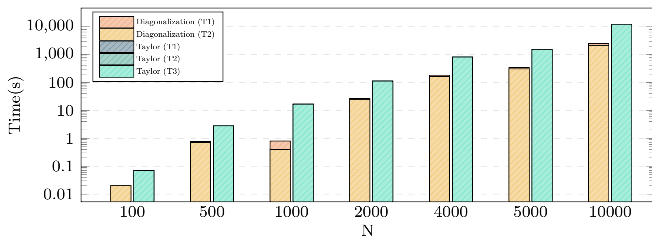
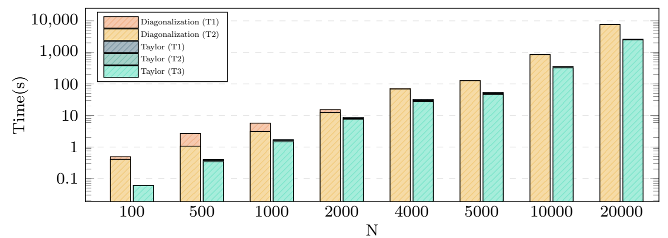


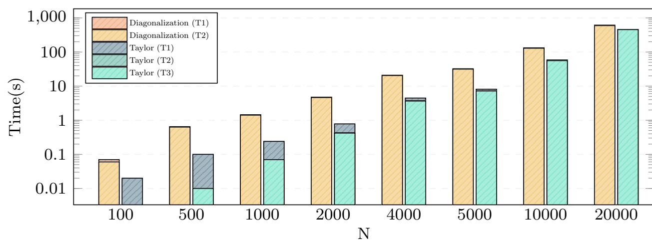
Figure 2: NVIDIA RTX 2080 Ti die[3]



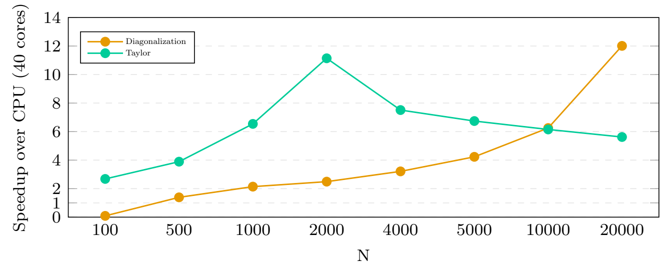
(a) Execution times on Jean Zay (CPU Version) (Using 1 core)



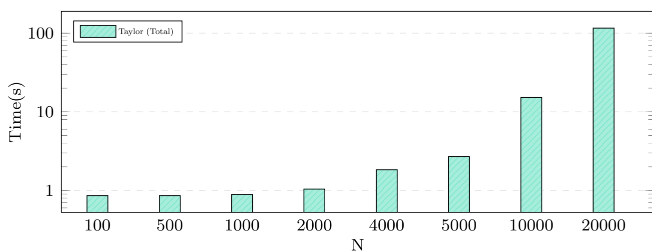
(b) Execution times on Jean Zay (CPU Version) (Using 40 cores)



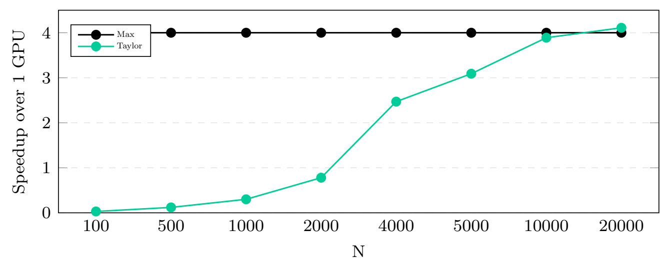
(c) Execution times on Jean Zay (GPU Version)



(d) Speedup obtained comparing CPU version (using 40 cores) and GPU version (using 1 GPU)



(e) Execution times on Jean Zay (multiGPU Version) (Using 4 GPUs)



(f) Speedup obtained comparing GPU version against multiGPU version (Taylor series only)

In this project, we first attack the matrix exponentiation doing a CPU version. As we know, GPUs are more powerful in this kind of scenarios. Therefore, after the CPU version, we make a GPU one. We will see that the performance gain is pretty decent. Finally, we decided to make a version that uses multiple GPUs inside a computer (in the case of Jean Zay supercomputer, we have 4 GPUs on the same computer!).

Jean Zay Supercomputer

The Jean Zay supercomputer[4] is a tier-1 machine and based at IDRIS. It contains a CPU and a GPU partition. One node of the CPU partition consists of 2 Intel Xeon Gold 6248 processors with 40 cores in total and its frequency is at 2.5 GHz. Whereas, one node of the GPU partition has 4 NVIDIA V100 SXM2 GPUs with 32 GB. It has an accumulated peak performance of 28 PFLOP/s.[4]

Matrix exponentiation

In the *deMon2k* software package the calculation of the RT-TDDFT (Real Time-Time Dependent Density Functional) is based on Magnus propagation. This involves the calculation of the exponential of a complex matrix without any special properties.[5] In order to compute the matrix exponential, there are different algorithms:

- Matrix diagonalization
- Taylor expansion
- Baker-Campbell-Hausdorff

We will make implementations following two of them. Each one will be detailed in the following sections.

Taylor series

The expression of the Taylor series applied to the exponential is:

$$\exp(A) = \sum_{n=0}^{\infty} \frac{A^n}{n!} = \frac{A^0}{0!} + \frac{A^1}{1!} + \frac{A^2}{2!} + \frac{A^3}{3!} + \ldots$$
$$= Id + A + \frac{A^2}{2} + \frac{A^3}{6} + \ldots$$

We are working with matrices, so in the expression, A is the input matrix. Thus, in Taylor's expansion, we need to compute A^x (which is done by matrix multiplications), divide A by a given value, and add the partial sums. The expansion is infinite, but, in the practice, we have to make it finite. Here appears the concept of iterations. Instead of calculating an infinite sum ($\sum_{n=0}^{\infty}$), we fix the number of iterations. For example, 50 ($\sum_{n=0}^{50}$). This means that the matrix exponentiation computed by Taylor expansion is an approximation. However, for the *deMon2k* project, the exact value for the exponential is not needed, and a number of iterations of 50 is enough to achieve the needed precision.

Even though there are many mathematical operations involved in this computation, in practice, the matrix multiplication takes almost the 100% of the computing time, since the matrix multiplication algorithm is $O(n^3)$, against the sums, factorial or divisions, which are $O(n)$.

Diagonalization

The calculation of the matrix exponential $\exp(A)$ using diagonalization contains two steps: At first an eigenvalue decomposition of the matrix A is per-

formed,

$$A = Q\,D\,Q^{-1},$$

where Q contains the eigenvectors and D is a matrix with the corresponding eigenvalues stored at the diagonal. Afterwards, $\exp(A)$ can be computed easily by

$$\exp(A) = Q\,\exp(D)\,Q^{-1},$$

where $\exp(D)$ is a diagonal matrix containing the exponential of the diagonal entries of D. The computationally most expensive part of the algorithm is the calculation of the eigenvectors and -values.

Implementation

CPU

The implementation in CPU uses functions of the LAPACK[6] (Diagonalization) and BLAS (Taylor) libraries. Specific subroutines are shown in Table 1, along with the timer's name we used to measure each of these functions in our benchmarks. This way, we can understand better where are we spending the execution time and how to improve it.

	Taylor	Diag.
$T1$	cblas_zaxpy	zgesv
$T2$	memcpy	zgeev
$T3$	cblas_zgemm	

Table 1: Functions for the CPU implementation whose execution time is measured

For benchmarking on 40 cores, the Intel Math Kernel Library (MKL) is linked because it includes optimized LAPACK routines taking advantage of several cores. The benchmarking results are obtained using the following compiler and library versions:

- Compiler: GCC version 8.3.1
- Intel-MKL version 19.0.4

and are shown in Figures 1a and 1b. We measure the execution times of different parts of our code as well as the total execution time as an averaged value over 3 runs. Taylor series runs for 50 iterations. As we can see, diagonalization achieves better results than Taylor for 1 core. However, their roles change if we use the 40 cores (this is, we fully use both CPUs). This winner exchange infers an interesting fact: Taylor scales much better than diagonalization, since for 1 core is worse, but if has the possibility of using many cores, it does a more efficient usage of them.

GPU

The implementation on GPU uses MAGMA[6] and the naming convention of its routines is very similar to the LAPACK. The `magma_zgeev` function is a hybrid CPU/GPU function to calculate the eigenvalues and -vectors of A. Furthermore, the functions `magma_zgesv_gpu` and `magma_zgemm_gpu` are used to compute the inverse matrix and perform matrix multiplications. Special care has to be taken since the last two functions require the input data stored on GPU.

The following environment is used for benchmarking:

- Compiler: GCC version 8.3.1
- CUDA version 10.2
- cuBLAS version 10.2
- MAGMA version 2.5.3

magma_	Taylor	Diag.
$T1$	zaxpy	zgesv_gpu
$T2$	zcopy-matrix	zgeev
$T3$	zgemm	

Table 2: Functions for GPU implementation whose execution time is measured

The functions used for the GPU version are shown in Table 2 and the benchmarking results are visualized in Figure 1c. Both implementations become faster compared to the implementation on the CPU as can be seen in Figure 1e. In total, Taylor performs better but this is not surprising because matrix multiplication is very efficient on a GPU and the data transfer is minimized.

Multi GPU

A multi GPU version means that the computation takes place in more than a GPU at the same time. In our case, we use 4 GPUs in parallel. However, we only did a Taylor version for multi GPU. Moreover, instead of starting from the base of the GPU version, the multi GPU version has been implemented using cuBLAS and CUDA. Therefore, we did a new version with cuBLAS. After that, we added support for multi GPUs, by deciding how to divide the work among the different GPUs involved in the computation. The benchmark results are depicted in Figure 1e and 1f. With large input matrices, the speedup obtained against the single GPU version is optimal (4x). Thus, we can conclude that our multi GPU version is quite efficient. In fact, computing the exponentiation of $N = 20000$, almost the largest matrix that fits into GPU memory, takes less than two minutes.

Conclusions

The matrix exponentiation is an operation that is very well suited to be performed on a GPU. Both algorithms showed a significant speedup compared to their corresponding version on a CPU. Anyhow, the calculation using Taylor expansion is faster than the one using diagonalization.

References

[1] http://www.demon-software.com/public_html/index.html [Last accessed 29 August 2020 - Online]

[2] https://en.wikichip.org/wiki/File:kaby_lake_(quad_core)_(annotated).png [Last accessed 25 August 2020 - Online]

[3] https://tweak.de/grafikkarte/msi-geforce-rtx-2080-ti-gaming-x-trio [Last accessed 25 August 2020 - Online]

[4] http://www.idris.fr/eng/jean-zay/index.html [Last accessed 28 August 2020 - Online]

[5] Wu, X et al "A. Simulating electron dynamics in polarizable environments" J. Chem. Theor. Comput. 2017, 13, 3985–4002

[6] http://performance.netlib.org/lapack/ [Last accessed 28 August 2020 - Online]

[6] https://icl.cs.utk.edu/magma/ [Last accessed 28 August 2020 - Online]

PRACE SoHPC Project Title
Matrix exponentiation on GPU for the deMon2k code

PRACE SoHPC Site
Maison de la Simulation (CEA/CNRS), France

Pablo Martínez

PRACE SoHPC Authors
Pablo Martínez, University of Murcia, Spain.
Theresa Vock, Vienna University of Technology, Austria

Theresa Vock

PRACE SoHPC Mentor
Karim Hasnaoui, France

PRACE SoHPC Contact
Pablo Martínez
E-mail: pabloantonio.martinezs@um.es
Theresa Vock
E-mail: theresa.vock@tuwien.ac.at

PRACE SoHPC Software applied
LAPACK, BLAS, MAGMA, cuBLAS, LaTeX, git

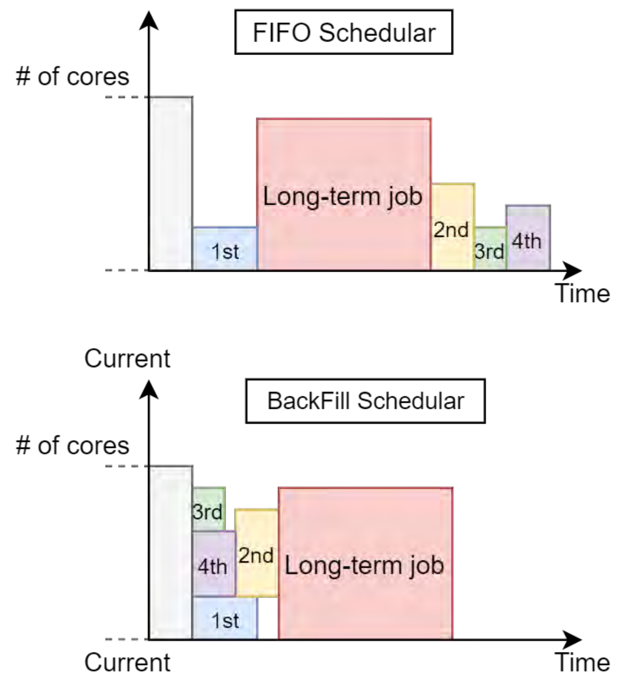
PRACE SoHPC More Information
deMon2k webpage

PRACE SoHPC Acknowledgement
We would like to thank Karim Hasnaoui for all of the support and help he has provided us since we started working on the project. It was much easier to solve the problems we found in our way thanks to his experience and knowledge on the field.

PRACE SoHPC Project ID
2014

Predicting Job Run Times

Francesca Schiavello, and Ömer Faruk Karadaş



Cluster users tend to give large over-estimations of the time needed to run their jobs. This over-estimation can grossly impact the scheduling of these jobs in a negative manner. Using machine learning we aim to more accurately predict these times, whilst avoiding under-estimations.

Supercomputers are very important tools for performing the calculations needed in the development of scientific studies. Some of the jobs that are accelerated using supercomputers are completed quickly and some can continue for months. Since each user cannot have their own supercomputer, these computers must be shared. Therefore, a workload manager is needed for the clusters to efficiently handle management of the submitted jobs. To solve this problem, SLURM workload manager is a useful and practical tool.

SLURM has several optimizations to schedule submitted jobs most efficiently. The optimization we focus on in our study is given by backfilling. Unlike the standard first in first out (FIFO) scheduling, backfilling allows jobs to be completed earlier by distributing short-term jobs on idle nodes of supercomputer.

However, some people can declare limit times for their jobs which are much longer than the actual elapsed time (Figure 1), which makes the scheduling optimizations ineffective. Our project proposes a solution to this problem using machine learning with various mathematical approaches.

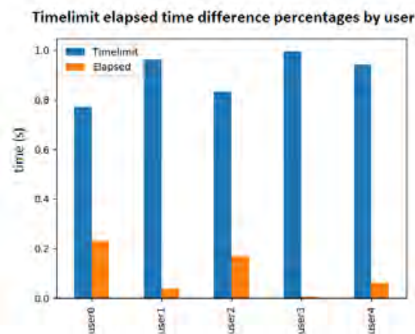


Figure 1: Normalized Elapsed time and Time limit ratio of submitted jobs

Additional info

- Machine learning is a good starting point when it comes to determining time limits in the place of users doing it and lots of data is required to train a good machine learning model. The dataset consists of the past batch jobs of the supercomputer of Hartree Centre.
- As a result of the analyses made, we have seen that similar jobs generally elapsed for similar periods. Therefore, grouping similar job names would make it easier to estimate the time limits. Levenshtein distance (edit distance),[4] one of the famous methods in natural language processing, was used when grouping job names. To summarize briefly, Levenshtein distance determines how many edits (insertion, deletions, or substitutions) are required to convert one word to another. Groupings were made with 2 or less edit distance
- While determining the accuracy of the algorithm, we established

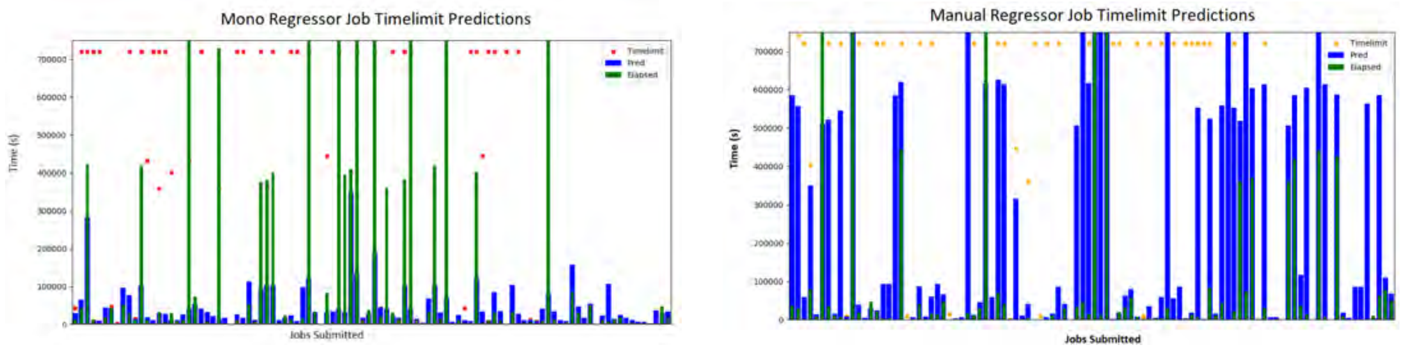


Figure 2: Predictions made with Mono (Huber regression) on the left and manual regression on the right.

that predictions longer than the elapsed time and shorter than Time limit were correct predictions.

Job Time limit Regression

When estimating the time limit, the prediction should not be less than the elapsed time. In case of underestimation, people's days of work can be wasted, because their submitted jobs would be terminated prematurely, forcing them to submit their jobs again.

Our aim here is to create a model that estimates the elapsed times of the jobs according to the job names and determines the time limits with a certain safety margin.

Mono Regressor

As a first approach, we chose to create a model using regression algorithms already existing in the scikit library, and we called it a mono regressor. Various regression methods from the scikit library were used, and the highest performances were obtained with Gradient boosting, Random Forest and Huber regression, but there were still many underestimates. As seen on the left in Figure 2, the weakness of this model is that each regression model has a general attitude, so it cannot predict outsider jobs that do not conform to the general attitude in the train set.

Manual Regressor

As a second approach to reduce the number of underestimations, we developed a regression model that can be characterized manually and we called it manual regressor.

In the Manual regressor, data are grouped by job names and the maximum elapsed times of each job names are selected. To avoid underestimations, predictions are made above a certain margin of safety from the maximum elapsed time (e.g. %150).

In addition, the manual regressor chooses not to predict the jobs that are elapsed at a certain rate close to the time limit. It also chooses not to predict the jobs that are less than a certain frequency (e.g. 3). As seen on the right in Figure 2, the weak point of this model was that it could not predict a job that elapsed more than the maximum time in past jobs.

Machine learning - a cyclical process

While the results obtained from the previous models are already good, and the model itself is powerful because it stores all of the user's history, this latter fact would result in re-training the model quite often. This is because each submitted job with a unique enough name becomes a new parameter, as such the algorithm must re-learn all the previous relationships and the new relationships due to this new parameter. To avoid problems due to this, and because Machine Learning is in fact a cyclical process, we can take what we have learnt in the previous model and apply it to a less expensive one.

Through the prior testing we know that both the history of elapsed time and past job names, contain a lot of predictive power. To determine their relevance to the model we run a feature importance test. This again can be done easily through Python's scikit learn package. This test ranks our features by order of importance, and we find that previous job elapsed times, and the similarity between job names are some of the most important predictors, which conforms with our expectations. We also find, unsurprisingly, that as we move backwards in time, the importance of each job decreases, that is to say, that the last job is more important than the 2nd-to-last, which in turn is more important than the 3rd-to-last, and so forth. Through testing and research [2], we choose to focus on the last three previous jobs, with the addition of adding features for the max, mean, and standard deviation for elapsed times of the previous 10 jobs. Adding more will likely add unnecessary noise to our model, while adding fewer could lose information on the variance.

Regression Techniques

Using the features we have explained, we can train a model on a partial subset of the data we have and test it on an unseen subset of the data, this is needed to evaluate the performance of each model. We initially tested a multitude of models but found that Linear Regression models and Ensemble method models perform best overall. By testing on multiple, disjoint subsets of the data, we can achieve a more true view of the performance of the models. Given that our data is time-dependent we cross-validate by training and testing on ordered chunks of our data, rather than randomly sampled ones like in non time-dependent cases.

It is worth mentioning that although Random Forest, and another tree ensemble method achieved high accuracy rates on their own, we chose to combine these methods to decrease the number of under-estimations even further. Note that in this case accuracy refers to predictions being greater than the actual elapsed time, if the prediction were to be below, then users would have their jobs terminated by the scheduler prematurely. As many users aren't used to check-pointing - saving their results periodically - all their work would be lost and they would have to resubmit their job at the back of the queue. By combining the different regression models, we take the maximum estimated time of three different models, thus effectively reducing the number of under-estimations.

Despite this effort, a small percent-

age of under-estimations remain. With our best efforts these remain around 7%. This is also partly due to human error, where users sometimes may themselves give the scheduler a time limit for their job that is too low, in such a case, the algorithms will likely come up with estimates that are under-estimates as well (though there are a few times where the algorithm will suggest a higher time limit). Mostly, these jobs are unpredictable and our regression techniques cannot estimate them accurately. As such, our best shot is to predict what jobs will fail in our regression model, and simply avoid estimating their run time.

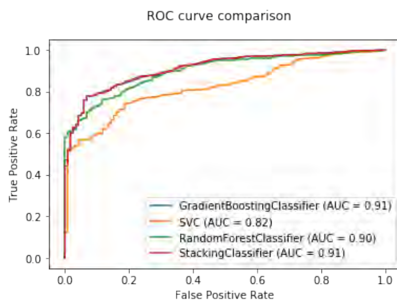


Figure 3: The ROC curve of various classification models. Notice random forest achieves the highest true positive rate (almost 60%) with 0 false positives.

Predicting the Unpredictable

After building a regression model we move on to build a classification model. In the latter we aim to classify jobs within two categories: good jobs, which are the job's run times that we can safely estimate without under-predictions, and bad jobs, which are exactly those jobs that are under-predicted and that would be terminated. Now, recall that our regression models predict most jobs accurately, so we have a ratio of about 93 good jobs, to 7 bad jobs. A simple model, trained on the current skewed dataset, would learn this imbalanced ratio, and give more importance to classifying good jobs, than to accurately classifying bad jobs. Gaussian Naive Bayes, which is a simple yet powerful model, simply classified all of our jobs as "good jobs" - hey, with that tactic it still achieved 93% accuracy!

The whole point of this second model though is to identify the "bad jobs", so our skewed dataset really won't cut it. To let the algorithm know that it should give more importance to bad jobs, we perform down-sampling, where essentially we train the model on a subset of data containing a balanced ratio (1:1) of good jobs to bad jobs.

If you are familiar with machine learning you may know that this method is heavily used in the medical field. If for example you were testing people for a rare, but deadly disease, you would much rather identify sick people at the cost of misclassifying healthy people, rather than the other way around. If your model were to achieve an astounding 99% accuracy, but not identify a single sick person, and just classify everyone as healthy - well then that would be a useless model!

This brings us to the second technique used to identify bad jobs. In a classification model with only two categories, our model can estimate the probability that a job falls in either category. This probability, by default, has a threshold of 50%, such that a job is classified in the category for which it has higher probability of falling into. We can change this threshold to be skewed such that a job is not classified as good unless it has 60%, 70% or even more of a probability to be a good job. With a high enough threshold this trick allows us to identify all bad jobs, without exception. In our case, we found through multiple testing that this threshold is about 70%. This of course comes at the cost of missclassifying good jobs. A cost function is usually used in such cases to determine how many jobs we are willing to missclassify, but first we must identify the cost of missclassifying each respective category. Such a cost function would give us the adequate threshold to use.

In figure 3 we can see that the classification model performs quite well. The area under the curve, AUC, which is a better measurement to use rather than accuracy for this scenario, has a best value of 0.91, where the maximum score would be 1, and it would form two straight lines pulled up at the top left corner. A diagonal line from (0,0) to (1,1) is equivalent to what a random prediction model would achieve.

Future work

In our project we have built two models for the estimation and classification of jobs, to predict more accurate run times without causing any under-predictions. This is a new approach in the literature, as no researchers, to the best of our knowledge, have tackled the issue of avoiding under estimating jobs. Like any machine learning algorithm, there is always room for improvement. A number of directions are possible for future work. The first thing would be the live testing on the system, such that we could calculate the actual efficiency gained in the scheduling algorithm. It would then be possible to study what kind of jobs are the ones that would create the most improvements in scheduling, and focus on predicting those job run times more accurately. We could fine tune the algorithms parameters and weights to achieve this. The techniques could then be influential on the SLURM scheduling parameters chosen, to optimize the scheduling algorithm even more. Finally we could expand the classification categories to include and predict jobs that were under-estimated by the user. By predicting these we could suggest to the user a more appropriate increased time limit, such that their job would not be terminated prematurely. This would again save a lot of the cluster's resources and diminish the queue's waiting time.

References

[1] Gaussier, Eric, et al. "Improving Backfilling by Using Machine Learning to Predict Running Times." Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis on - SC 15, Nov. 2015.

[2] Naghshnejad, Mina, and Mukesh Singhal. "A Hybrid Scheduling Platform: a Runtime Prediction Reliability Aware Scheduling Platform to Improve HPC Scheduling Performance." The Journal of Supercomputing, vol. 76, no. 1, 2019, pp. 122–149.

[3] Tanash, Mohammed, et al. "Improving HPC System Performance by Predicting Job Resources via Supervised Machine Learning." Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines (Learning), 28 July 2019.

[4] Levenshtein, Vladimir I. "Binary codes capable of correcting deletions, insertions, and reversals." Soviet physics doklady. Vol. 10. No. 8. 1966.

PRACE SoHPCProject Title

Machine Learning for the reschuduling of SLURM jobs

PRACE SoHPCSite

Hartree Centre, UK

PRACE SoHPCAuthors

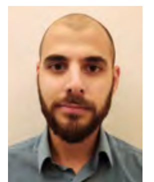
Francesca Schiavello, University of Amsterdam, Netherlands

Ömer Faruk Karadaş, Izmir Institute of Technology, Turkey

Francesca Schiavello

PRACE SoHPCMentor

Vassil Alexandrov, UKRI STFC, UK



Ömer Faruk Karadaş

PRACE SoHPCAcknowledgement

We would like to acknowledge the great help we have received from our project Co-Mentor, Anton Lebedev, in guiding and pushing our work to achieve the goals we had initially set forth.

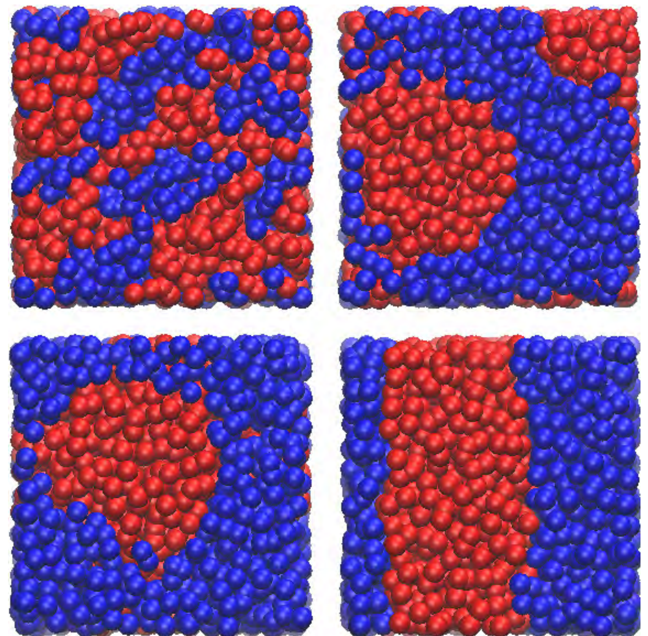
PRACE SoHPCProject ID

2015

Porting Lowe-Andersen and Peters thermostats to GPU for a particle simulation program DL_MESO

Boosting Dissipative Particle Dynamics

Nursima Çelik & Davide Crisante



In particle simulations, thermostats are used in controlling system temperature, like in real life experiments. In this project, we ported Lowe-Andersen and Peters thermostats to GPU, enabling researchers to use these thermostatting options in reasonable amounts of time.

Conducting scientific experiments on computers gives us a chance to explore more variety of systems in less time. We are able to replicate many real world phenomena on computers and get more insight about mechanisms of nature.

Evidently, it is important to have simulations that produce correct outputs. We don't want our "replication of nature" to deviate from reality and lead us some crazy conclusions. To get correct physics, one thing we need to pay attention is the scale that simulation operates in.

As you may have heard, things work a little different in micro compared to macro scale. While Newton physics is sufficient in defining rules of the world in macro, quantum effects become significant when we go down to the micro level. For this reason, it is good to have different simulation techniques for different scales.

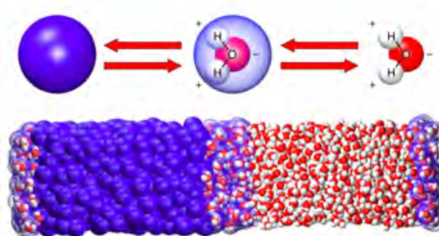


Figure 1: Group of atoms are treated as one particle in DPD.

The scale we are concerned in this project is mesoscale, which is in between micro and macro. Mesoscale can range between 10 –1000 nm and 1 ns –10 ms[1]. One technique used for this scale is called Dissipative Particle Dynamics (DPD).

A glance at DPD technique

In DPD, particles are modelled as spheres, and time is split into small durations called time steps. There may be forces acting on particles which will cause motion. At every time step, we calculate total force applied on each particle, and then we find its next position and velocity. The output is trajectory of every particle as well as system statistics like kinetic/potential energy, temperature, pressure.

In any experiment, it is desirable to keep some quantities (like temperature, pressure, volume) constant to observe the effect of independent variable. Mechanisms for keeping temperature are called thermostats. It is important for simulations to be able to provide thermostats.

Task

There are two versions of DL_MESO code, one written in Fortran language, utilising MPI; the other written in CUDA-C language, utilising GPU. While there are five different types of thermostats in Fortran, only DPD thermostat was available in CUDA. Our task was to port Lowe-Andersen (LA) and Peters thermostats from Fortran to CUDA.

DPD Thermostat vs Lowe-Andersen vs Peters

DPD thermostat is the default option, which is a balance between drag and random forces. Drag forces tend to decrease the temperature by decreasing velocities, and random forces tend to increase it by reproducing the Brownian motion.[2]

On the other hand, LA and Peters thermostats introduce a velocity correction step after integration instead of drag and random forces. LA and Peters thermostats are similar but yet different. LA applies velocity correction only to a sample of particle pairs instead of all pairs. Specific equations of methods are omitted, but they can be found in the DL_MESO User Manual.

Methodology

We took Fortran code for Lowe-Andersen/Peters options as our reference. In every step, it was important to make sure that both versions produced the same results when fed with the same inputs.

We divided code into small steps so that we port one step at a time. Those steps included:

- Velocity Verlet Stage 1
- Force Calculation
- Velocity Verlet Stage 2
- Velocity Correction
- Stress and Kinetical Component Calculation

In order not to get lost during the process, we followed a workflow that looks like as follows.

- Make sure values from both versions matches before the implementation
- Implement the step
- Make sure values from both versions matches after the implementation

We often encountered with mismatched results from serial and parallel versions. Problem was sometimes an uninitialised variable, but sometimes it was not that easy - like one we faced during velocity correction.

Now let's take a closer look to each step.

Step 1: Velocity Verlet Stage 1

Stage velocities of all particles through the half time step.

The default DPD thermostat was already using a kernel called *k_moveParticleVerlet_1* for this purpose. Since Lowe-Andersen/Peters had no difference in this step, we used the same kernel in our Lowe-Andersen/Peters integration.

Step 2: Force Calculation

Between two Velocity Verlet stages, forces should be calculated.

A particle applies force to all particles around it within a determined distance called cutoff radius. Also, it is affected by only particles those are in the cutoff radius. As effects of particles beyond this radius is considered to be small, their contributions are ignored.

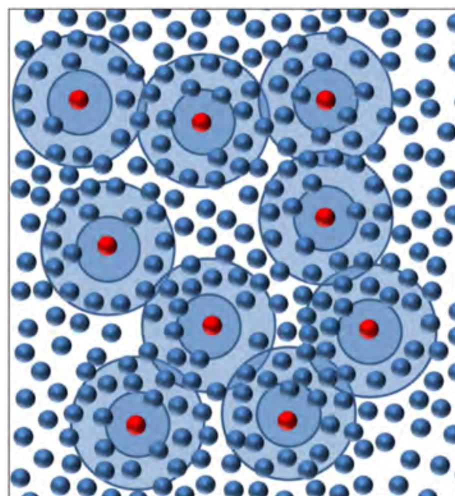


Figure 2: Particles interacting in cut-off radius

In this step, we loop through all particles to find pairs that are interacting with each other. As the result, we get total force applied on each particle at the end.

As you see, force calculation may become a heavy process; especially when the number of particles and number of time steps is large. In the results section, we will see that force calculations indeed take a lot of time. It is the most costly second operation.

To realise this step, we modified a kernel called *k_findForces* that was used for DPD thermostat. We removed random and drag forces.

Step 3: Velocity Verlet Stage 2

Stage velocities up to the end.

This is computationally similar to stage 1. Again, we made use of a kernel called *k_moveParticleVerlet_2* that was used in DPD thermostat. But we needed to make a modification.

Inside this kernel, stress and kinetic energy calculations were done, which was normal for DPD thermostat. However, in Lowe-Andersen and Peters, there is a velocity correction step after Velocity Verlet, in which velocities of particles are updated. If we calculated stress or kinetical values in this step, they would no longer be valid after the correction.

So, we created *k_moveParticleVerlet_2_lowe*, removing stress and kinetic energy calculations. We left them to post correction.

Step 4: Velocity corrections

As we mentioned, in DPD thermostat, random and drag forces have the key role to keep temperature around the same level. Here is the innovation of Lowe-Andersen/Peters.

In Lowe-Andersen we needed to

1. Get a random sample of particle pairs
2. Adjust their relative velocities according to a Maxwellian distribution[3

For the case of Peters, we needed to

1. Get all particle pairs

2. Adjust their relative velocities according to a Maxwellian distribution[3

As correction was new, there was no kernel we could use as we did for Velocity Verlet. We created a kernel called *k_correctLowe/k_correctPeters*.

To apply velocity corrections, we needed to get all interacting particles.

In the serial version, the interacting particle pairs were stored in a list while calculating forces. In the correction stage, that list is used to avoid re-traversing all those pairs. The pair list was shuffled. Then in a loop new velocities were calculated and assigned to each pair.

In GPU version, we could make a list of interacting pairs in the force calculation phase, too. But this would cause extra memory transfers, from device to host, and again host to device - something we wanted to avoid.

Therefore, we decided to find pairs once more, this time in order to correct their velocities. Although this is not the most optimal thing to do, it has the advantage of simplicity.

At first, we corrected only a controlled group of particles. That way, we were able to compare the updated velocities from CPU and GPU.

Data Race

In CUDA implementation, data race caused trouble.

Imagine one thread is on the way to updating pair *(1,2)*, while the other wants to update pair *(2,1)*. Finding new velocities depend on the difference between current velocities of two particles. When one particle finds correction and updates the velocities of *particle 1* and *particle 2*, the velocity difference in the other thread becomes invalid. So, the correction the second thread adds ends up being incorrect.

Incorrect values was a minor problem, because modifications were small. However, the big problem was randomness. Values differed from one execution to other, depending on the order of writes of threads. That made impossible to compare GPU values with those of CPU.

To prevent data race, one option was to use atomic operations. But that would mean to serialise the code. Instead, we made threads to use old velocities instead of current ones. It didn't matter which thread modified velocity first, because nobody read those updated velocities in the correction phase. All threads used the ones before correction.

As a conclusion, we did not strictly obey the formula of thermostats. But, as I mentioned already, this corrections are small enough to allow us to do such a simplification.

Step 5: Stress and Kinetic Energy Calculation

We finally added stress and kinetical component accumulators. This step was important in the sense that we could see if temperature stayed around a determined value as intended.

Results

We tested final program with Lowe-Andersen and Peters options both in CPU (32 cores) and GPU. GPU version turned out be 10 times faster.

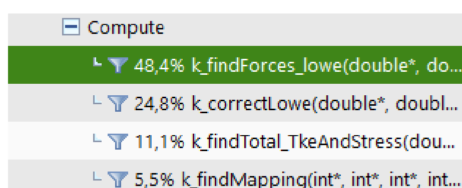


Figure 4: First three most costly functions after memory transfers

We used nvprof program to profile the code. The most costly operation is memory transfers from host to device. Thankfully they happen only once at the beginning of the program. In the second and third places, there are force calculation and velocity correction functions - remember both contained a traversal of all particle pairs which is costly.

Figure 3: On the top LA, on the bottom Peters: CPU time is in blue, GPU time is in green.

When we compare Lowe-Andersen and Peters, we see that the former takes less time when number of time steps are equal. Remember that velocity corrections were applied to a smaller group of particles in Lowe-Andersen. However, Peters has the advantage of allowing one to use larger time steps.

Discussion & Conclusion

We have two new features added to the GPU version of DL_MESO program: Lowe-Andersen and Peters thermostats. We hope that researchers will benefit having these options on GPU. As future work, other thermostats can be ported to CUDA version, like DPD-VV and Stoyanov-Groot. In additon, scaling DL_MESO to large number of GPUs would make a great impact in simulation of larger systems.

References

[1] Michael A. Seaton , Richard L. Anderson , Sebastian Metz & William Smith (2013) DL_MESO: highly scalable mesoscale simulations, Molecular Simulation

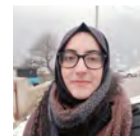
[2] Robert D. Groot and Patrick B. Warren (1997) Dissipative particle dynamics: Bridging the gap between atomistic and mesoscopic simulation. Journal of Chemical Physics, 107(11):4423-4435

[3] M. A. Seaton and W. Smith DL_MESO USER MANUAL

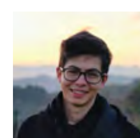
PRACE SoHPCProject Title
Scaling the Dissipative Particle Dynamic (DPD) code, DL_MESO, on large multi-GPGPUs architectures

PRACE SoHPCSite
Hartree Centre - STFC, UK

PRACE SoHPCAuthors
Nursima Çelik, Bogazici University, Turkey
Davide Crisante, University of Bologna, Italy



Nursima Çelik



Davide Crisante

PRACE SoHPCMentor
Jony Castagna, Hartree Centre - STFC, UK

PRACE SoHPCSoftware applied
DL_MESO

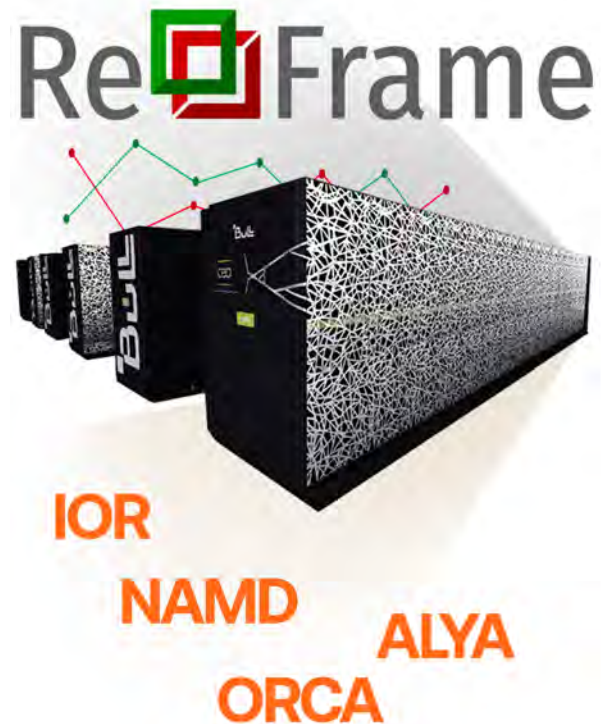
PRACE SoHPCMore Information
www.scd.stfc.ac.uk/Pages/DL_MESO.aspx

PRACE SoHPCAcknowledgement
Many thanks to our project mentor Jony Castagna for all the help and patience, and Hartree Centre STFC for the resources.

PRACE SoHPCProject ID
2016

Monitoring HPC Performance

Elman Hamdi, Jesús Molina Rodríguez de Vera



The variety of competitive hardware solutions has made benchmarking an increasingly significant and challenging task for HPC specialists. In this project, we used portable automating frameworks to better understand the performance of several HPC components and applications on different hardware and software configurations.

The complexity of HPC systems is constantly growing, and nowadays they are very heterogeneous environments:

- There are different hardware solutions, with multiple processors and accelerators architectures.
- Different parallelisation approaches are possible, like pure distributed memory, shared memory, hybrid or GPU.
- Moreover, there is a wide variety of scientific software offered to the system users, and each program uses the resources in a different way.

This heterogeneity can be a problem not only for HPC maintainers, that have to ensure that everything works smoothly and fast on their systems; but also for users, who don't know how to get the best performance out of the system for their application.

In order to overcome this issue, HPC specialists use a series of automating tools and frameworks to monitor the different hardware and software components of the system and get the best possible performance without sacrificing maintainability. Figure 1 shows the main tools and the workflow used at SURFsara:

- XALT, that is used for software usage monitoring. It helps maintainers to know which applications are used the most.
- EasyBuild, which makes the management of the modules of an HPC system much simpler.
- ReFrame, that is a high-level framework for writing regression tests for HPC systems.
- Jenkins for continuous integration of software and automatic test triggering.

All of the mentioned tools played an important role in our project.

Figure 1: Automated workflows used for software stack deployment on SURFsara.

However, we will focus on ReFrame, which has been our main working tool during this internship.

ReFrame allows creating abstract and portable regression tests, separating the logic of the tests from the low-level details like the hardware or configuration. ReFrame tests are written in simple Python code. Its syntax allows to easily modify parameters such as the EasyBuild modules to be loaded, the system partitions to be used (that is, the type of computation nodes), and even the desired parallelisation configuration.

Monitoring Cartesius

The goal of this project is to use the ReFrame framework to automate the testing of specific software and hardware components of Cartesius, the Dutch national supercomputer.

File systems

File systems play an important role in the performance of a supercomputer. It is thus crucial to make sure that the performance of these file systems is reliable and consistent.

In order to help maintainers to monitor this key aspect of the system, we created a series of ReFrame tests that could be run periodically to detect potential issues that may affect the performance of the system.

We used IOR, a synthetic benchmark commonly used for evaluating the performance of parallel file systems, to test the different file systems available on Cartesius (NFS, Lustre).

Heavily used applications

At SURFsara, XALT is used to monitor software usage and decide on which modules to focus to have the most impact.

This way, we identified NAMD and ORCA as two of the most heavily used applications on Cartesius, which makes it crucial to test them thoroughly to be able to determine:

- whether the performance conforms to the expectations and is consistent after maintenance and new installations, or immediately identify changes affecting performance,
- whether the performance and scalability are satisfying with all hardware and software configuration available,
- what configurations the users should choose to get the best performance out of the system for their specific application.

In order to answer these questions, we created ReFrame tests and ran them with different parallelisation configurations (pure MPI, pure OpenMP and hybrid) on the different hardware available on Cartesius: Intel Haswell, Ivy Bridge and Broadwell nodes, and Nvidia Tesla GPUs.

The results of our tests allowed us to identify and fix issues affecting current installations, and gave us meaningful insights regarding performance. With this information, we created extensive guides that are published on SURFsara's user documentation to help users to determine which hardware and software configurations they should use depending on their needs to obtain the best performance out of Cartesius. This, in turn, will help users make better use of the resources, increasing the throughput of the system.

Moreover, our tests will be used to make sure new installations perform at least as good as previous installations, and also to make sure that changes in the systems do not affect the application's performance in a negative way.

Figure 2 shows a simplified diagram of the workflow for this part of the project.

ORCA

ORCA is a general-purpose tool for quantum chemistry with particular emphasis on open-shell spectroscopic properties, parallelised with MPI. We used version 4.2.1 of ORCA, with OpenMPI 3.1.4.

We selected a test case performing geometry optimisation of a [Fe(H2O)6]3+ molecule employing DFT with RI approximation.

To measure performance, we used the total execution time in milliseconds(msec). Figure 3a shows the execution time on one Broadwell, Haswell, and Ivy Bridge node using different numbers of MPI tasks per node. This test shows good scalability up to 6 tasks per node, and then degraded scalability for higher core counts. The test case is too small to scale to a full node, and the overhead of the parallelisation is too high compared to the compute work when the number of tasks increases.

More details about the results of our analysis on Cartesius will soon be available on the ORCA userinfo page of SURFsara.

NAMD

NAMD is a parallel molecular dynamics code designed for high-performance simulation of large biomolecular systems. We used version 2.13 of NAMD.

We selected test cases of different sizes (number of atoms) and with different types of systems to be more representative of all real use cases. The metric we considered was ns/day, which is the most common for NAMD benchmarking. It represents the number of nanoseconds of simulation time per day of wallclock time.

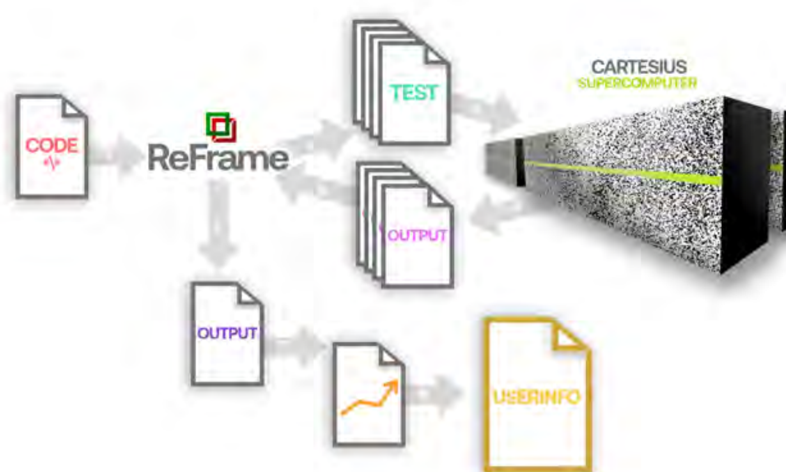
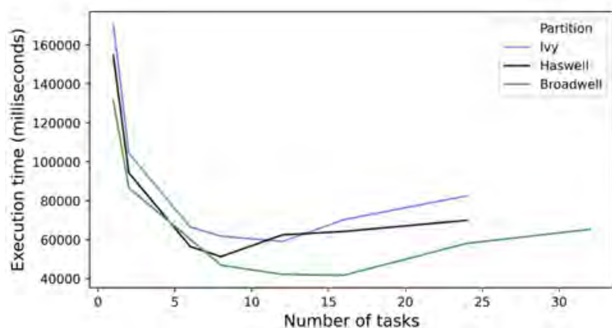
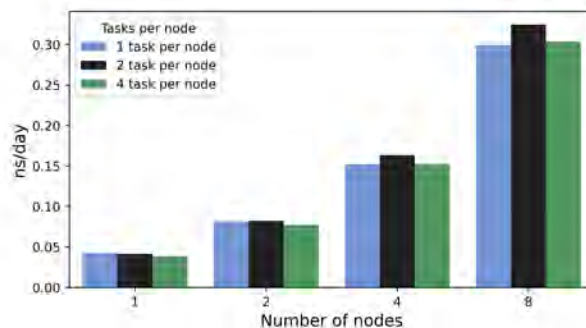


Figure 2: Workflow of test execution and analysis for ORCA and NAMD.



(a) Performance results depending on the number of MPI tasks for [Fe(H2O)6]3+ Molecule Test Case, ORCA 4.2.1



(b) Performance results depending on the number of MPI tasks for UEABS Test Case A, NAMD 2.13 Memopt, Hybrid MPI+OpenMP, Broadwell nodes

Figure 3: Examples of the results of our analysis of ORCA and NAMD performance.

With our tests, we noticed that the multithreaded version of NAMD was not available in the existing installations. Thus, we improved the Cartesius software stack by installing a memory-optimised version of NAMD that enables the usage of multithreading and allows larger experiments by reducing the amount of required memory. We compared the performance of this new module (using different hybrid parallelisation combinations) with the already existing module. No significant differences in performance were measured between the standard and the memory-optimised versions, which is good as it means users can run larger tests without losing performance.

As an example, we can see in Figure 3b the performance of the memory-optimised version on Broadwell nodes for the UEABS test case. The scalability is good when increasing the number of nodes, and the best parallelisation configuration is using hybrid MPI+OpenMP with 2 MPI tasks per node (i.e. 1 MPI task per socket).

More details about the outcomes of our analysis can be found in the NAMD userinfo page of SURFsara that we created as a result of our work.

Collaborating in the CompBioMed project

We also tested and analysed the performance of Alya, which is a multi-scale, multi-physics simulation code, and is part of the CompBioMed European Centre of Excellence.

The Alya developers of the BSC were very interested in understanding how a new feature of their program (a new version of their library Dynamic Load Balance, DLB) behaves on Cartesius, so we collaborated with them to obtain meaningful metrics from the execution of Alya. For that, we used a respiratory system simulation provided by BSC.

We based the analysis of the performance on the application of the methodology proposed by POP European Center of Excellence. We particularly focused on comparing the computation efficiency and load balance of the global application and the Nastin computation module using different heterogeneous hardware combinations.

The latest Alya version was not fully installed on Cartesius at the time of our participation in the project, so we could not run all the desired tests. However, we could extract meaningful results for some aspects of the execution that are being used by SURFsara and BSC specialists. Moreover, we created tests for the remaining parts so that they could be used once everything is ready in the system.

Profiling

We performed a performance audit of Alya using MAQAO, a performance analysis and optimisation framework recommended by the POP Centre of Excellence that operates at binary level with a focus on core performance. In order to integrate its usage in the SURFsara workflow, we wrote a step by step guide on how to use MAQAO to perform a performance assessment following POP methodology[1] .

Conclusions

The results of our work during this Summer of HPC 2020 are already being used for helping the HPC community, including maintainers, users and developers. In addition, thanks to the portability of the created tests, they will be used for future monitoring of the SURFsara systems.

References

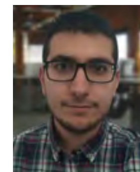
[1] POP Centre of Excellence in HPC. *How to create a POP performance audit Version 1.1*. https://pop-coe.eu/sites/default/files/pop_files/whitepaperperformanceaudits.pdf

PRACE SoHPCProject Title
Benchmarking and performance analysis of HPC applications on modern architectures using automating frameworks

PRACE SoHPCSite
SURFsara, The Netherlands

PRACE SoHPCAuthors
Elman Hamdi İzmir Institute of Technology, Turkey
Jesús Molina Rodríguez de Vera University of Murcia, Spain

Elman Hamdi



Jesús Molina Rodríguez de Vera

PRACE SoHPCMentors
Maxime Mogé, Sagar Dolas and Marco Verdicchio
SURFsara, The Netherlands

PRACE SoHPCContact
Maxime, Mogé, SURFsara
E-mail: maxime.moge@surf.nl

PRACE SoHPCSoftware applied
ReFrame, EasyBuild, XALT, Jenkins, IOR, ORCA, NAMD, Alya, MAQAO

PRACE SoHPCMore Information
reframe-hpc.readthedocs.io, easybuilders.github.io/easybuild, xalt.readthedocs.io, jenkins.io, ior.readthedocs.io, orcaforum.kofo.mpg.de, www.ks.uiuc.edu/Research/namd, www.compbiomed.eu/services/software-hub/compbiomed-software-alya, www.maqao.org

PRACE SoHPCAcknowledgement
PRACE and the organizers of Summer of HPC.
Our mentors at SURFsara.

PRACE SoHPCProject ID
2017

TSM of HPC Job Queues

Cathal Corbett, Joemah Magenya

The project goal is to create a monitoring system to capture real-time information regarding the number of jobs running on HPC clusters, while integrating the work with DevOps and Continuous Integration practices and tools. The information regarding the status of the job queues will be collected, processed and stored as time series and summarised and made available to users and administrators in the form of graphs.

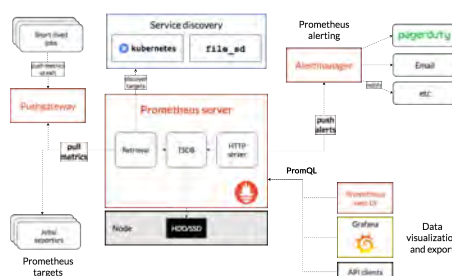
Time series analysis plays a crucial role in real time based systems like supercomputers. It allows users to be able to detect anomalies within their systems. It gives an overview of how a system behaves for a given period of time. A user can take advantage of time series monitoring to analyse trends of HPC based systems. Basically, through the use of time series, one can be able to extract information such as when a cluster node has stopped working (anomaly detection) and how many tasks are running including counting failed or suspended tasks.

Our project consists of configuring a Time Series Monitoring system which collects data on job queues, process the incoming data and a displaying the data on job queues in a timely and visually appealing manner.

Throughout the development of the software, DevOps and Continuous Integration tools and practices are employed. The GitLab CICD pipeline and runners will be utilised for the project to automatically build, test, and deploy source code written. Through the use of DevOps, code and testing errors are automatically detected and this allows the developer to correct and submit clean code as part of the final product.

Additionally, the use of Infrastructure as a Code (IaaC) tools and software will aid in the seamless configuration and deployment of the project.

The Project - TSM System



The project was split into two major tasks, configuration & deployment of the TSM system and secondly, statistical analysis of job queue data to verify any time series properties that may exist.

The Prometheus architecture diagram gives a very good overview of what our Time Series Monitoring system looks like. The architecture is divided in three main components.

1. The Prometheus Target is a http endpoint where a script is being executed which creates metrics from information produced by a job or service running and makes these metrics available over the above http endpoint.

2. The Prometheus Server makes a pull request from the Prometheus Target http endpoint for these metrics created by the executing script. The metrics scraped are stored as Prometheus' own flexible query language called PromQL.

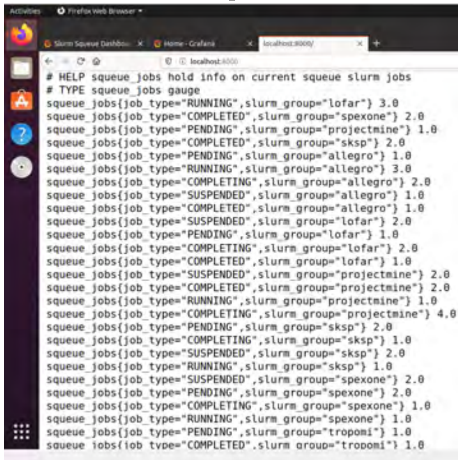
3. Data Visualization of these metrics is created by executing a pull request of metrics stored as PromQL from the Prometheus Server. The resulting

PromQL can be parsed and refined by the end user and displayed on graphs.

Job Queue Exporter

The Job Queue Exporter acts as the Prometheus Target. Information we are interested in is job queues of HPC schedulers. Two main schedulers exist for HPC systems: Slurm & TORQUE. Because Slurm is the most well known and used scheduler, the rest of the report will focus on the Slurm squeue command which returns information on job queues for the Slurm scheduler.

It was decided to make the first implementation of the exporter in Python3 due to more programming experience with Python before converting the exporter to the preferred Prometheus exporter language of Golang. The Python3 script running executing the squeue command, retrieves information on job queues running, parses the incoming information extracting only relevant fields, converts the essential information into a Prometheus Gauge Metric and makes the metric available over http://localhost:3000.



The exporter has been created as a Python pip package and can be installed through pip3 install job-queue-exporter. Unit testing of the exporter used pytest which is Python's testing package. The tox tool automates and standardises testing in Python and is used in conjunction with pytest. Testing of the exporter is deployed within the GitLab CICD pipeline where tox testing of the exporter is initiated using a Python3 docker image.

Unfortunately, the learning curve to implement the exporter in Golang required more effort. However, having full knowledge of how the exporter works made a seamless conversion from writing the exporter in Python3 to Golang. The Golang exporter package can be installed by executing go get followed by the Git repository of the source code. This alone makes the Golang exporter a preferred tool as you do not have to update an external package manager such as PyPi for Python with an updated version of the package every time a new commit is made to the Git repository. With Golang, the source code and package manager is hosted within the one location, Git.

Prometheus Server

Fortunately, the Prometheus Server takes care of all the scraping of the exporter http endpoint, processing of metrics and storing of information in PromQL which leaves very little to worry about for the developer.

The number one file in configuring Prometheus is prometheus.yml which specifies all the Prometheus Targets, their associated http endpoint where metrics are exposed and scraping time interval. Prometheus is running by default at http://localhost:9090/.

Other information can be configured in the prometheus.yml associated with Prometheus Rules and the built-in Prometheus AlertManager system. One rule configured for our system is when a Prometheus Target experiences downtime, the system maintainer can automatically be notified over email or Slack and can take the correct measures to revive the failed node.

Grafana

The Prometheus Web UI can be used to visualize Prometheus PromQL queries. However, Grafana is a more elegant and stylish data visualization software that is compatible with Prometheus. Two tasks need to be completed before getting nice dynamic graphs of job queue information.

1. Configure the Grafana DataSource to pull from Prometheus at http://localhost:19090/prometheus/.
2. Configure the Grafana dashboard which is stored as JSON data.

Our graphs are created with PromQL queries in the following format: squeue_jobs{job_type=<job_type.name>} which creates graphs for individual job queue types. The graphs can be further inspected by clicking on an individual {{slurm_group}} in the graph legend. System administrators of the system can now view job information in visually appealing graphs. Ansible - Full System Deployment

Ansible is the open-source software provisioning, configuration management, and application-deployment tool enabling infrastructure as code that fully automates deployment of the above system with the click of a button. In addition to running the above tasks, the Ansible playbook configures Prometheus and Grafana to run behind a reverse proxy.

Often, machines have a limitation, due to firewall policies or other security reasons, only opening to the external world ports such as 80. If you take a closer look at the Prometheus Targets running on the server, it is evident that Prometheus is actually running over port 19090 instead of 9090 and Grafana over port 80 instead of 3000.

Time Series Analysis (TSA)

Time series analysis comprises methods for analysing time series data in order to extract meaningful statistics and characteristics of the data. TSA allows to forecast the trends of HPC clusters through the help of Prometheus and Grafana platforms. The data is feed to prometheus and grafana servers, the time series analysis is done by taking advantage of the build in Prometheus and Grafana dashboards which allow to visualise the data in the form of graphs. Our analysis based primarily on three aspects of time series which are seasonality, autocorelation and stationarity paying attention only to the RUNNING jobs of spexone, allegro and sksp slurm_projects. The allegro slurm_group showed some seasonality characteristics. Seasonality refers to periodic fluctuations.

The trends in the spexone project where autocorellated, that is there were similarities between observations as a function of the time lag between them.

Grafana Data Visualisation Ansible Deployment

Stationarity is one of the importance characteristics of time series, if the observations don't change over a period of time they are said to be Stationary, thus the same trends are observed.

Results – What did I find out?

Cathal: The result is fully functioning code that automatically configures, deploys and runs the Time Series Monitoring system consisting of the Python3/Golang Exporter, Prometheus Server and the Grafana Data Visualisation tool. The exporter has been primarily tested and deployed for the Slurm scheduler. However, it would be easy to adapt the exporter to function for other HPC schedulers such as TORQUE by identifying the job queue information format of the scheduler and modifying the parser to suit that format.

Apart from becoming knowledgeable on Prometheus and Grafana tools, the following results were achieved: From developing the exporter, my standard of Python programming has improved immensely from being immersed in Python packaging, testing using pytest and tox, entry points, Python standards, coding conventions and documentation. Being able to translate the exporter to Golang introduced me to a whole new language and discovered similar features to the Golang language as listed above. Additionally, using DevOps and Infrastructure as a Code tools and practices had developed me into a more experienced and competent computer scientist now capable of tackling all stages of the project and producing better, more reliable and tested code. Tools such as GitLab CICD & runners and Ansible have showed me the power of being able correctly configure and deploy production code. My knowledge of Linux has been expanded through the configuring of background systemd services. Virtual environments needed to be configured for this project to run the full system in an isolated environment.

Joemah: After having been involved in all the parts of the monitoring process, from the acquisition of the metrics, transmission, storage and visualisation, I developed an interest of working within an HPC environment. My programming skills were taken to another level as I learned knew skills like CI/CD and DevOps. Moreover, I become more knowledgeable with working within a virtual environment as well as executing commands on the linux shell confidentially. I got a chance to grasp knowledge to working with Prometheus and Grafana platforms. In addition,I learned the importance of working as a team and not being shy to asking questions. In shot, TSM project boosted my abilities and confidence to meeting a specific goal.

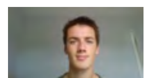
Discussion & Conclusion

In conclusion, a faculty with HPC systems with an interest in deploying a Time Series Monitoring system and displaying information on job queues can easily deploy such a configuration utilising the Ansible script developed. The Ansible script is available on the GitLab repository and full installation instructions are explained in the README file.

If system admin of these faculties have a different HPC schedulers configured to the above Slurm default, the source code of the exporter can be amended to parse the data produced by that different scheduler.

PRACE SoHPCProject Title

Time Series Monitoring of HPC Job Queue



PRACE SoHPCSite

SURFsara, Netherlands

Cathal Corbett, Joemah Magenyaphoto

PRACE SoHPCAuthors

Cathal Corbett, Joemah Magenya, NUIG, Ireland, Padova, Italy

PRACE SoHPCMentor

Juan Luis Font, Ansterdam, Netherlands

PRACE SoHPCContact

Name, Surname, Institution
Phone: +12 324 4445 5556
E-mail: leon.kos@lecad.fs.uni-lj.si

PRACE SoHPCSoftware applied

Prometheus, Grafana, Gitlab CICD, Ansible, Golang, Python3, Slurm, TORQUE, Linux, Ubunutu, Virtualbox.

PRACE SoHPCMore Information

https://summerofhpc.prace-ri.eu/time-series-monitoring-of-hpc-job-queues/
https://gitlab.com/surfprace/cathal

PRACE SoHPCAcknowledgement

Acknowledgements goes to Juan Luis Font and the SURFsara Spider team for the support provided throughout the project.

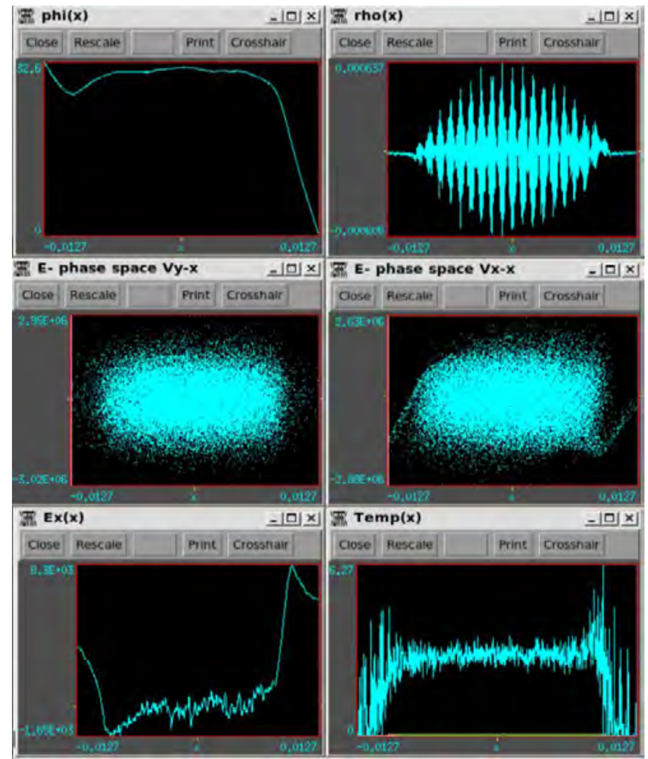
PRACE SoHPCProject ID

2018

Improving the performance of plasma kinetic simulations using GPUs

Accelerating Particle In Cell Codes

Víctor González Tabernero, Shyam Mohan



Particle In Cell (PIC) codes are at the forefront of modern plasma kinetics and nuclear fusion research. However, plasma kinetic simulations can be extremely time-consuming as millions of particles are simulated at once. The power of GPUs and heterogeneous computing can be harnessed to run these simulations faster.

Plasma is the rarest state of matter found in nature. It is composed of charged nuclei and electrons at extremely high temperatures where the electric and magnetic fields dominate the behaviour of matter. We can find plasma in many applications such as nuclear reactor cores, rockets, lasers etc. We are interested in plasma kinetics of **nuclear fusion reactors**, which might be the next revolution in energy resources. These reactors are still in development stage and due to their high cost, (e.g. the nuclear reactor Tokamak from ITER costs around 24 000 million euros) scientists and engineers cannot conduct enough tests to develop a functional core.

To minimise costs, computer simulations of the reactor are performed to get the best description of the reactor's behaviour. In this project we simulate the kinetics of plasma which forms one of the main parts of the core that needs to be understood in depth. The simulations involve many interesting aspects from mathematics, physics and computation such as mathematical modelling based on the **kinetic model** description of plasma. This model requires solving multiple equations to determine the position and other physical properties of each *super-particle* (a bunch of actual particles in plasma). This technique is called ***Particle-in-cell*** (PIC) simulation and it is commonly used for this.

Particle in Cell Simulation

PIC simulations provide a description of the plasma's properties based on the behaviour of its particles which are simulated individually (in small bunches of particles). To do so, time is discretized and the equations that describe the behaviour of particles are solved for each time step. The common features of a *PIC* code that are taken into account each time step are:

1. **Particle mover**: updates position and velocities of the simulated particles according to the famous Newton's laws of motion. The velocities are affected by the fields generated by all the particles.
2. **Field solver**: calculates the fields inside the simulated spatial region at some grid points. There are two kinds of solvers: the electrostatic ones that only calculates electric fields and electromagnetic solvers that calculates electric and magnetic fields. These solvers use the information of charge density distribution, charge fluxes and reactor features. All these interwoven quantities are governed by a set of coupled partial differential equations called Maxwell's equations
3. **Accuracy of the simulations improvements**: *PIC* codes can also include more features to provide a better description of reality. For example they can simulate multiple atomic species, reactor injection of particles, particle decays in boundaries, particle collisions, particle reactions etc.

There are many detailed articles and reports about different *PIC* codes. You can find more information in the article[1] or in https://www.particleincell.com/.

Simple PIC - SIMPIC

In our project, we focused on `SIMPIC`[2] (Simple PIC) which is a **simplified *Particle in cell* code**. The `SIMPIC` code was developed under certain hypotheses which make the simulation significantly easier.

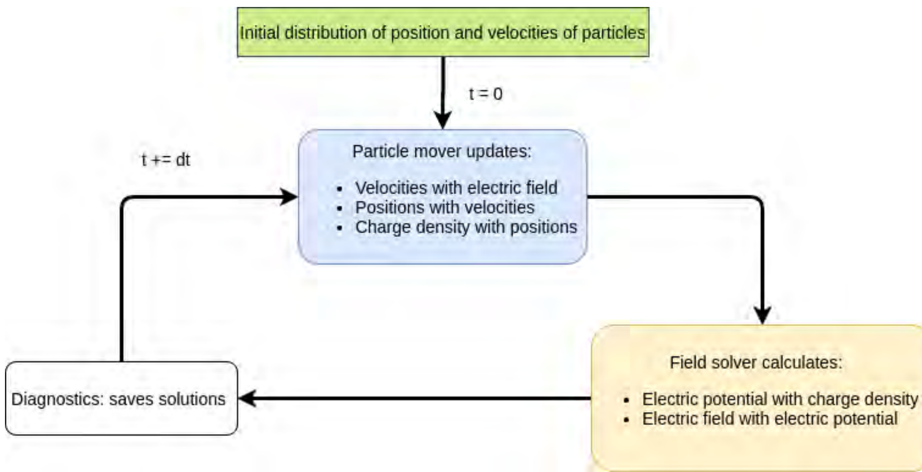


Figure 1: `SIMPIC` workflow diagram. Shows the general algorithm flow of this code.

There is assumed to be no collisions between particles, no magnetic field and only free electron particles (no ions). As per these assumptions, the complicated Maxwell's equations boils down to solving only a Poisson equation for the potential. This is easily done using the well-known finite difference method. Now, the field can be calculated simply by taking gradient of the potential. In a PIC code, the whole plasma region is divided into subregions called cells. And inside each of these cells, there are some particles (ions/electrons). We give an initial random distribution of particles inside the plasma device. Then, we apply an external electromagnetic field to these particles usually in the form of a voltage source. After this initialization, the PIC codes follow a common algorithm as seen in Figure 1 for `SIMPIC`.

GPU Parallelization of SIMPIC

GPUs perform computations at a much faster rate than CPUs. However, they cannot handle other parts of the code, like branches or conditional statements very well. Hence, if we can offload the computationally expensive parts of our code to a GPU while running the rest of our code on a normal CPU, we can expect some speedup in our code. This is what we tried. We created a GPU version of the particle mover and field solver functions. This was done using CUDA programming for GPUs. How does this parallelisation work? We can imagine a GPU as a CPU but with a huge number of cores in it. Each core in a GPU can work independently on each loop iteration, for example. Obviously, this would be much faster than running it on a single core in a CPU in a sequential manner. However, the important criteria for GPU parallelization is 'independence'. Each GPU must be able to work on a loop iteration independent of other threads. Otherwise, we might get wrong results. Keeping this in mind, we tried to parallelize the two main computations in our code: the particle mover and the field solver.

Particle Mover

Before we can move the particles, we need to know what forces are acting on each particle. This force is derived from the surrounding electric field. However, this is not so trivial. It is important to understand that cells in the plasma region form a grid and the potential and electric fields are calculated only at these "grid points". The particles obviously can be anywhere inside each cell. Hence to find out the exact force acting on the particle we perform an interpolation process called 'gathering'. We take the field values from the two boundaries of the cell in which the particle belongs, and interpolate them to the exact position of the particle. A typical particle mover algorithm would be something like:

1. Gather field at particle position
2. Calculate new velocity using field
3. Calculate new position using new velocity

As can be seen in the particle mover algorithm, each thread can now be assigned to a single particle as each particle moves independent of any other particle. Hence, it is a very 'parallelisable' algorithm. One should also note that the GPU has its own memory space. Before any computation can be done on it, the required memory needs to be transferred to the GPU. This transfer is generally the bottleneck for a GPU application. You can imagine that the transfer of the position and velocity data of a million particles will take a lot of time. Hence, we decided to create the particles in the GPU alone to avoid this memory transfer. We also implemented an optimised algorithm for particles that go beyond the plasma region using a Boolean array to flag particles alive/dead. This aids in vectorised processing in GPUs.

Field solver

For the field solver, we can find a difficulty on this parallelization and it comes from the discretization of the equations. To calculate the electric potential, the code has to compute the solution of a tridiagonal matrix system which comes from the Poisson's equation, and this is a very sequential calculation. There are many algorithms designed to do this calculation, but CUDA comes with a library called cuSPARSE for algebraic calculations in GPU. This library contains a function which calculates the solution for this tridiagonal system. The parallelization process for the rest of computations of the field solver part is similar to the particle mover, but in this case, each thread is assigned to a grid point.

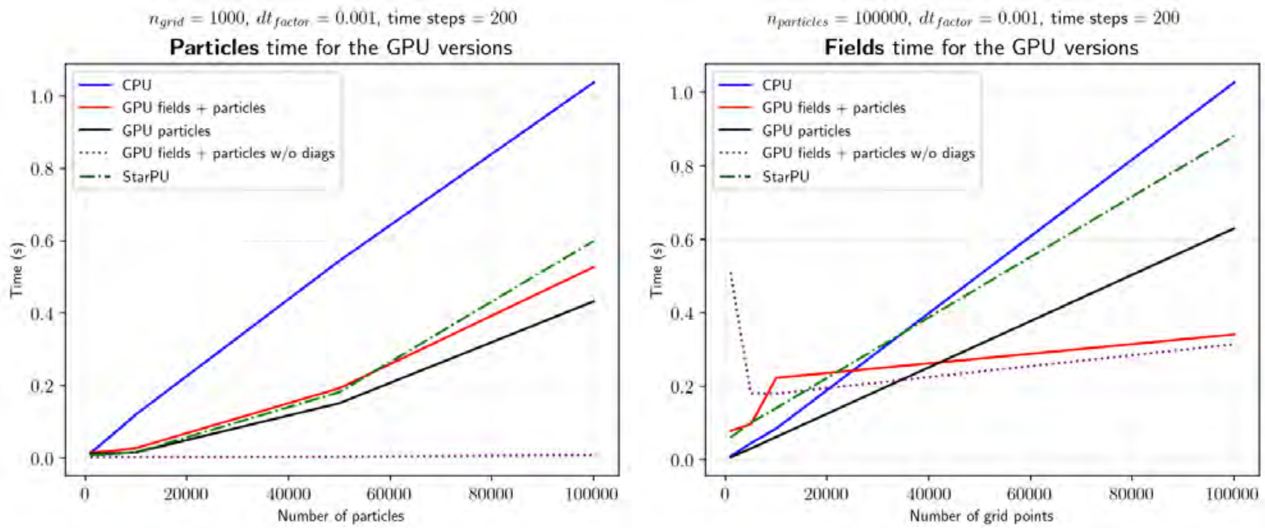
In summary, the parallelization of the field solver follows the next steps:

1. Solves the tridiagonal matrix using an external library.
2. Corrects electric potential with the boundary values.
3. Calculates the electric field with the electric potential.

We have to note that the two last steps are independent and, consequently, efficient for GPU calculations.

Heterogeneous Computing using StarPU

Now, we can make even better use of our computing resources if we can use both the CPU and GPU for computation. This is done by creating tasks or 'codelets'. These are just some additions to the existing code which assigns all the Processing Units (CPU/GPU) certain



Comparing performance of various accelerated versions of SIMPIC. Left: Runtime Plot of the Particle Mover against number of particles. Right: Runtime plot of field solver against number of cells. The NVIDIA Tesla K80 accelerator in the GPU node of VIZ Cluster, University of Ljubljana was used for this study.

tasks to execute: in our case, particle mover and field solver. This was done using StarPU, which is a software tool which can schedule tasks to run on heterogeneous architectures (CPU+GPU). All memory transfers and allocations in the application is done by StarPU itself, which saves some amount of code. Also, asynchronous tasks can be run on multiple processing units, with each task working on a data subset. All these possibilities were implemented on our field solver and particle mover functions.

Results and Conclusions

Our benchmarks of the SIMPIC versions show that the **GPU versions have much better performance than the CPU version**. The GPU particle mover shows a speedup of greater than 5x, which is to be expected as the particle calculations are well parallelised and the CPU-GPU memory transfers have been optimised. However, this speedup seems to saturate as we increase the number of particles above 10^5 particles.

In our code, the calculation of charge density is also included in the particle mover function. This density is calculated by extrapolating the charge of a particle located within a cell to both the grid points of the cell. However, this process is not very independent as different particles could add to the charge density at the same grid point. Hence, this calculation limits the performance of our code. With regard to this, it can also be observed that the number of particles per cell (PPC) affects the speedup. This could mean that the performance would be better if we have more number of cells for a given number of particles. On the other hand, the CPU-GPU memory transfer of the bigger density arrays associated with larger number of grid points also requires more time. Hence, we observed that there is an optimal number of particles per cell which would give us the best speedup for the particle mover.

On the other hand, for the field solver, we see that the GPU version is slower than the CPU one for low grid points, and it is faster for a high number of grid points. This time consumption mainly comes from the tridiagonal solver which is not efficient in GPU for low number of grid points but its calculation time remains more or less constant with the number of grid points. However, for the standard grid points that are used in this kind of simulations, which are usually low, **the best overall performance comes from the GPU particle mover and the CPU field solver**. We also have to note that the most time expensive part of this code is the diagnostic savings while also increases particle mover and field solver computations.

Our **StarPU version** of SIMPIC shows good speedup when compared to the CPU version. However, this is not as much as our CUDA-only GPU version. The particle mover runtimes for the StarPU version are slightly faster than the GPU version. This is because the StarPU data management is more efficient at data transfers. However, this advantage is negated when the runtime of the entire code is considered. This is because of the additional overhead of invoking the StarPU library and launching tasks. However, **StarPU enables portability of code**. It can be run on multiple architectures without any changes in code.

References

[1] D. Tskhakaya, K. Matyash, P. Schneider, and F. Taccogna, *The Particle-In-Cell Method*,Contrib. Plasma Phys. 47, No. 8 – 9, 563 – 594 (2007)

[2] SIMPIC code, https://bitbucket.org/lecad-peg/simpic/src/master/.

PRACE SoHPCProject Title

Implementing task based parallelism for plasma kinetic code

PRACE SoHPCSite

University of Ljubljana, Slovenia.

PRACE SoHPCAuthors

Paddy Cahalane, Dublin.
Shyam Mohan, Subbiah Pillai, Germany.
Víctor González Tabernero, Spain.

PRACE SoHPCMentor

Ivona Vasileska, UL, Slovenia.

PRACE SoHPCContact

Víctor, González Tabernero, University of Oviedo
Phone: +34 630 714 721
E-mail: vitagor@outlook.es
Shyam Mohan, Subbiah Pillai, RWTH Aachen
Phone: +0175 3469712
E-mail: s.mohan@rwth-aachen.de

PRACE SoHPCSoftware applied

C++, CUDA, OpenMPI, StarPU, Matplotlib

PRACE SoHPCMore Information

OOPD1, PRACE website, SoHPC website.

PRACE SoHPCAcknowledgement

We sincerely thank our mentor Ivona Vasileska and everyone from the LECAD laboratory for their valuable inputs and access to the VIZ Cluster. Special thanks to Prof. Leon Kos, the SoHPC coordinator, for his support and constant presence when we needed it

PRACE SoHPCProject ID

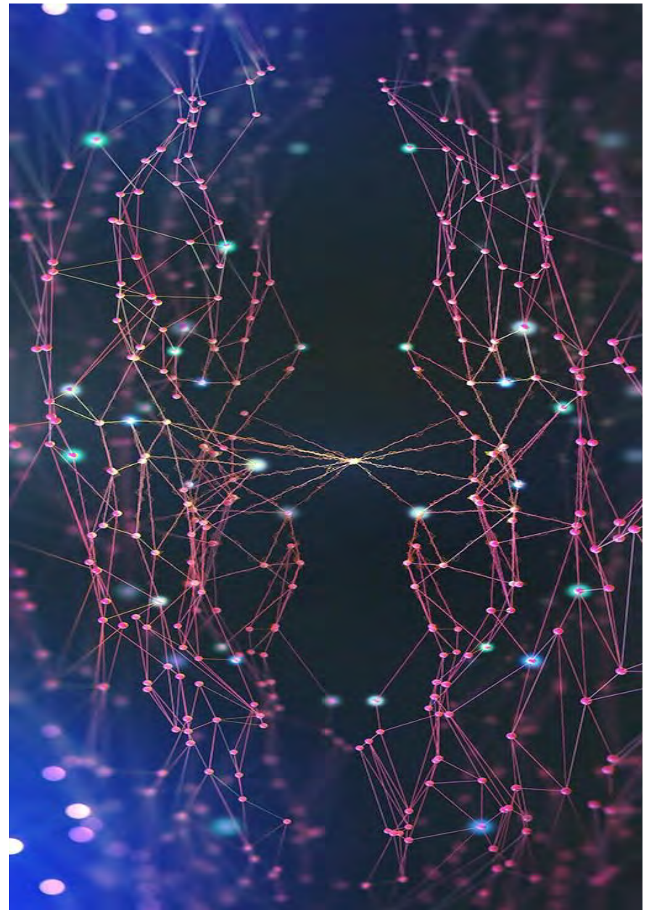
2019

Víctor González Tabernero

Shyam Mohan, Subbiah Pillai

When HPC meets integer programming

İrem Naz Çoçan , Carlos Alejandro Munar Raimundo



Mixing mathematics and computers could get a very good results in efficiency. It could obtain the solution of a very large problem that for a human being could be unfathomable in a few minutes. And now, imagine that you can do it even faster.

Branch and bound method is one of the most used methods for solving problems based on decision making. In the particular case that concern us, is a discrete case, this means that you have to decide either A or B.

In order to understand this let us explain the max-cut problem. Imagine that you have a weighted graph and you want to divide the vertices into two subgroups. After that, the connection between two nodes that are in different subgroups is cut. The goal is to bipartition the vertices so that the sum of the weights on the cut edges is maximal.

Turning back to the branch and bound algorithm, it is organised by a rooted tree. All starts with root node, where upper bound and lower bound is set. In every step it has a node to compute, and by updating the bounds and checking them it is decided to branch to another two nodes or prune because you will not get any further going through that branch. An Figure 1 there is an example of this. With this, you have fewer options to explore, and it will take less time to obtain the optimal solution to this problem.

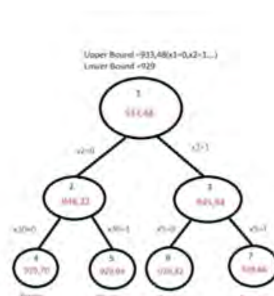


Figure 1: Diagram showing rooted tree of the branch and bound.

Notice that, we are working with max-cut problem that is a NP-complete problem, so we will need an heuristic to get an approximation to an optimal solution because getting the best solution could get even longer.

Nevertheless, we want thing going faster. As a consequence, this project came up. The aim of the project is to make exploration of the branch and bound tree even faster by concurrently exploring different branches. For that, we have done two different approaches: *master-worker approach*, and *one-side communication approach*.

In every modern computer, there is more than one core in a processor, so you can order every core to do some work. So, the processor is an office where workers go to do their jobs. Therefore, in master-worker approach there is one master process (load coordinator) who has all the information about the underlying graph and sends data and tasks to the workers. As a result, we change from a program that computes everything serial this means that all jobs are queued and done sequential; to a program that is more con-

trolled because all the jobs are assigned to workers. In addition, this will be executed in the supercomputer sited on the University of Ljubljana, Faculty of Mechanical Engineering.

Nonetheless, we will see that this will cause some idle time due to communication issues.

For that reason, we tried to code one-side communication. We thought that if we delete the communication between nodes by creating an area that can be accessed by any of the workers without needing a master, we could increase the efficiency of the program. We will see that this is not completely true; but as with everything, if it is not tested, it is not known if it will work. This area that can be acceded for everyone is called "window".

Used Methods

In the introductory part we have explained the project motivation, which is the problem to be handled and the way we will manage to get improvement of the code. From now on, we will explain both approaches in detail, their advantages and disadvantages and the results we have obtained.

Master-worker approach: The best way to understand something is to see something that you understand and compare both. So, let's make an analogy: imagine a very big company with a lot of people working there, roles are assigned to rank the responsibilities of each worker. As a result, simple scale, we have a director or manager and workers or employees. The manager assigns various tasks to employees, which fulfil them and/or send new ones to coworkers and the process continues in this way. Then these jobs are sent back to the manager to get them together.

Our implementation is based on this logic. One process is selected as master, which has all graph instances, and sends job requests to other worker processes. When the worker processes finish their work, they send the results to the master process and wait for the new task. Master process is responsible for controlling the entire process and keeps track of the status of each worker. The master creates a vector of the solution, which is 0 or 1.

To communicate between load coordinator and workers *MPI Send* and *Receive* functions are used. In Figure 2 is a diagram that reflects this. Master worker is at the middle and the workers are connected to master to send and receive information.

In our particular case, the nodes sent by the master to the workers are the ones to branch and evaluate, and the nodes sent back to the master are the evaluated children of this node. To do this we have used c structures for nodes in which we allocate: a fixing nodes (which are already on the solution vector); fractional solution which is used to get the next branch; the level of the tree where is being evaluated; and the upper bound.

The master determines which node will be sent to which process and when the results will be received. Therefore, idle time occurs while processes are waiting for to send back the work done.

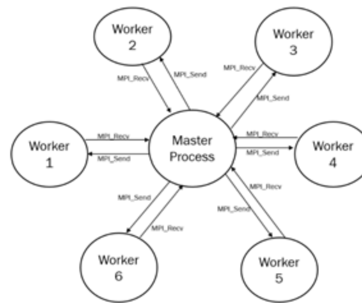


Figure 2: Diagram showing master-worker approach.

One-sided approach: Now that you are familiar to the company and you know well the roles that are assigned and the master-worker approach is well understood, let's restructure the company.

First of all, now there is no manager or director nor employees, everyone is responsible for its own work. Everyone is a manager and an employee at the same time. So, the idle time on processes waiting to send back the work done is no more a problem. You may think that this is like freelancers, and you are right; it is like a lot of people not related with the same porpoise: get the solution of max-cut probelm.

When one of the processes needs help by sharing one of its branch and bound nodes, other workers realise this and can get to its work directly, without waiting for the target process to send the message.

Accessing someone's work is done through the *MPI Window* allocated at the start of the algorithm. Note that this window is the determined area that other processes can reach and access data without involvement of the its owner. This section need to be implemented very careful to avoid race conditions. In our problem, this method is applied as shown in Figure 3.

All starts when process 0 branches the first node, and when it notices that it has more than one node in its queue, a free process reaches this node and evaluates it.

So, in this model, each processes can share its nodes with others. To get more points of view, we have developed two different versions for this access part shown in Figure 4.

As a consequence, we have version 1 which is the processes which are free inform to the others that are free, and allocates free nodes in his queue. But, version 2 processes finds which one is free and put the node in free process queue.

In order to send and retrieve data from the windows we used MPI Put and Get functions. However, if two processes access the same window at the same time it generates a race condition. This means that if they update the same value at the same time one of these values will be lost on the process.

Therefore, these operations needed to be executed in a way to prevent this situation. Managing this timing is what caused the idle time of the workers.

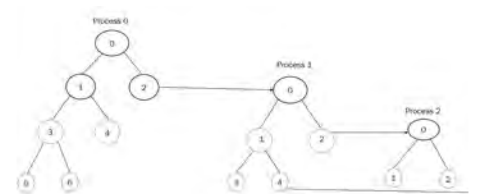


Figure 3: Shows the work of the one-sided approach.

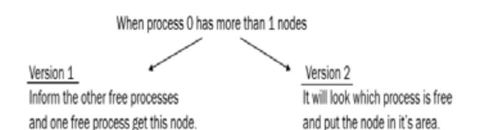


Figure 4: Two approaches for sharing information in one-sided communication.

Results

What we have done so far is to explain by facts the methods we have used and their behaviour. Nevertheless, these words mean nothing without some benchmarks to support them. Let's take a look at the numerical results of these approaches.

First of all, the benchmarks have been done by testing proposed parallel algorithms on 130 graph instances with a different number of vertices and edge weights. We took 5 instances for which the sequential branch and bound algorithm needed the longest time to obtain the optimum solution.

Notice that in Figures 5 and 6 ara tables displayed. These tables are the timing results of executing the master-worker approach and one-sided communication approach, respectively. All the graph instances contain 100 vertices.

These tables show the scalability results with both approaches when running the same instance with an increased number of workers.

As we can notice, from Figure 5 when increasing the number of processes the time decreases; being the serial version the one that takes longer times to solve the same problem.

Out of the number of running processes, one of them acts as a master process and the remaining workers evaluate the nodes. In other words, 1 master and 5 worker processes work in the case when 6 processes are assigned.

Process Numbers

3 prc	6 prc	12 prc	24 prc	48 prc	Serial
499.51	201.77	100.86	54.87	37.51	955.37
466.07	200.11	98.96	58.18	44.03	868.72
307.62	126.18	65.18	36.85	26.19	505.25
241.50	109.50	57.40	34.55	24.08	497.36
205.39	93.54	48.18	30.48	23.10	381.16

Figure 5: Reported timings of master-worker approach vs serial version.

Looking at the results of one-sided communication approach (Figure 6), occurs the same that in master-worker approach: we are getting a good improvement of the time that takes to solve the problem from serial to parallel version. Yet, they are not better than the results obtained on the master-worker approach.

3 prc	6 prc	12 prc	24 prc	Serial
603.09	420.36	244.83	144.58	955.37
555.70	416.58	283.15	165.99	868.72
348.91	271.99	141.50	95.71	505.25
310.18	210.45	145.50	87.25	497.36
256.95	171.44	123.53	72.60	381.16

Figure 6: One-sided vs Serial version

What's more, as a curious thing, we notice that if we use 48 processes or more the execution was like serial version. So, in that case we will not get any advantage of parallel communication when the workers increase. By way of comment, we think that this situation could be interesting to be study case.

Discussion & Conclusion

As we have been exposing, we proposed two different parallelization schemes of serial branch and bound algorithm, to solve a combinatorial problem. The first one is based on master-worker approach while the other utilises one-sided communication.

Looking at the results we can affirm that the parallelization was found to be successful, since the proposed algorithm could vastly reduce the computational time of the serial solver.

As we see, master-worker approach is more efficient when the problem is bigger, but when is not that big one-sided communication approach is also very useful.

It has been observed that, for the max-cut instances for which the optimum solution was obtained in short time, (i.e. the branch and bound tree is smaller) parallelization gives worse results as it requires more processing and more time is spent on communication. This means that if it is a small-sized problem is also suitable using the serial version; because the parallel version will not get much more advantage than serial.

Nonetheless, when we look at the five examples given in Figures 4 and 5, we see that the times are shortened in the two approaches compared to the serial version.

Although there is a constant decrease by looking at the tendency patterns, the results of the master-worker approach are better than the one-sided approach. The red line shows the result for the master-worker, while the blue line is the results of the one-sided approach. Results of the longest running instances were taken into account.

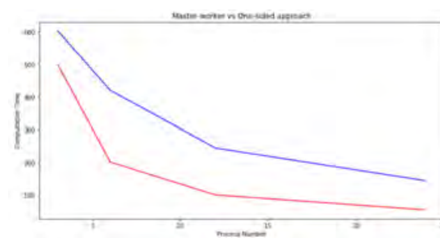


Figure 7: Graphical comparation between master worker approach and one-sided approach.

Despite the fact we thought that the one-sided approach would take less time than two-sided, the results did not come out as we expected. We think that the reason for this is that the processes need to do more control in order to prevent race conditions.

Acknowledgement

First of all,we'd like to thank our mentor Timotej Hrga for all support and interest through the two months, University of Ljubljana, Faculty of Mechanical Engineering for the use of their facilities. PRACE for giving us a place on the Summer of HPC programme.Many thanks to Leon Kos, Pavel Tomsic for all their work, in the first time online program they did everything well by organizing the training week, emails and weekly meetings .

References

[1] Franz Rendl, Giovanni Rinaldi, Angelika Wiegele. (2010). Solving Max-Cut to optimality by intersecting semidefinite and polyhedral relaxations

[2] Loc Q Nguyen. (2014). MPI One-Sided Communication

[3] William Gropp, Ewing Lusk, Anthony Skjellum.() Using MPI Portable Parallel Programming with the Message-Passing Interface

PRACE SoHPCProject Title
Implementation of Parallel Branch and Bound algorithm for combinatorial optimization

PRACE SoHPCSite
HPC cluster at University of Ljubljana, Slovenia

PRACE SoHPCAuthors
İrem Naz Çoçan , Carlos Alejandro Munar Raimundo

PRACE SoHPCMentor
Timotej Hrga, University of Ljubljana, Faculty of Mechanical Engineering

PRACE SoHPCContact
İrem Naz Çoçan,Dokuz Eylül University
E-mail: iremnazcocan@gmail.com
Carlos Alejandro Munar Raimundo, University of Almería
E-mail: caamura@gmail.com

PRACE SoHPCProject ID
2020

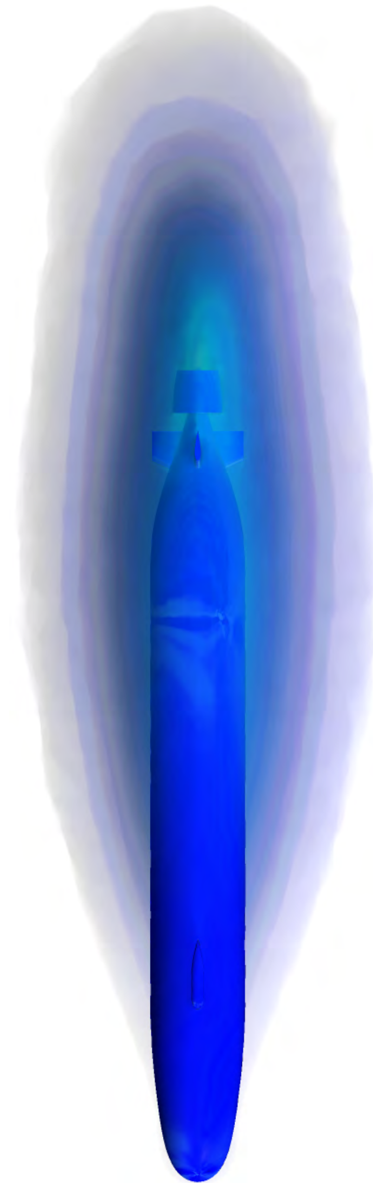
İrem Naz Çoçan

Carlos Munar

Investigating the effects of different turbulence models and drift angles on CFD simulations of the DARPA Suboff Submarine with HPC.

Drifting in a Submarine? Hold On!

Matthew Asker, Shiva Dinesh



Vehicle prototyping can be costly and time-consuming. Computational Fluid Dynamics (CFD) offers a faster and cheaper approach, allowing the initial stages of prototyping to be sped up massively. The main aim of this project was to test how different **turbulence models and drift angles** affect the drag coefficient and turbulent kinetic energy distribution on a submarine.

What is CFD?

Werner Heisenberg is attributed with once saying "When I meet God, I'm going to ask him two questions: why relativity? And why turbulence? I really believe he'll have an answer for the first".

Simply put, turbulence is a difficult physical phenomenon to model. So hard in fact, that we don't yet have the full solution. This is where the wonders of CFD come in. Although no exact solutions exist to our problems, we can use computers to approximate the problem and obtain solutions to a precision that is 'good enough' for us.

The project

In this project we used CFD methods to simulate the flow of water around the DARPA Suboff submarine, travelling at a speed of 6.5 knots. The simulations were taken for several drift angles (0-16°) and part configurations of the submarine.[1] We also utilised several different turbulence models for a particular configuration, to show the difference these models had on the drag coefficient and turbulent kinetic energy distribution.

How did we complete the project?

After we have established the main goals of the simulation, our first task is to create a model of the body around which the flow will be analysed. This usually involves modelling the geometry with a CAD Software package. There are two ways to model the geometry. Firstly, we can use the equations provided to describe the submarine and generate the model. The second method involves inputting the set of points provided into the ANSYS Design Modeller to obtain the model.[1] We have used the first technique to generate our model. The equations describing the model were input into a Python program and the model was generated using them.

Salome, free software with a python interface, has been used in generating the model. Now that we have generated the model, the extent of the finite flow domain in which the flow is to be simulated is to be decided.

To capture the full effects of turbulence, it is good practice to leave a space behind the submarine which is around 5-20 times the length of the submarine. In addition, it is good practice to leave a space of 2-5 times the respective length in other

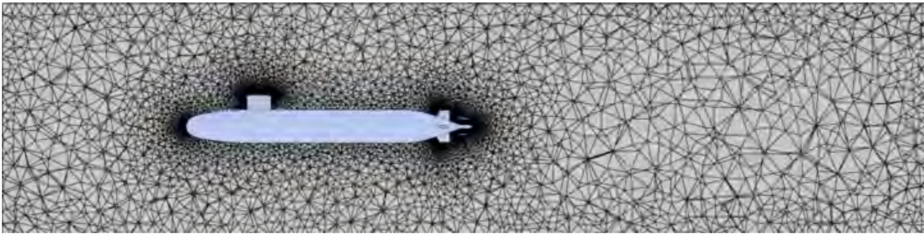


Figure 1: A basic mesh we used for some preliminary results.

directions. So, this finite flow domain would capture all the effects. The limits of this finite domain are the free boundaries out of which flow enters or exits. The surface of the submarine is also a boundary, through which the flow cannot enter or exit. The equation given in the reference paper[2] allows us to write Python code to create all the different parts of the submarine. We can then combine these parts as we please in the geometry.

The domain in between these boundaries is now required to be analysed. For analysing this, we need to discretise the flow domain into a grid. Here the continuous space is discretised into several contiguous finite volumes in a process called meshing. Meshing is one of the crucial elements of a CFD simulation.

There are several parameters which can be modified to fine-tune the mesh. We cannot discuss every parameter at length here, but the essence of meshing is to refine the mesh near the submarine surface to capture the turbulence generated due to it. This can be achieved by increasing the number of layers of mesh close to the submarine wall and reducing the size of the elements.

A dimensionless quantity y+, which measures how the distance between cells changes in the direction perpendicular to the surface, is a good indicator as to the quality of the mesh. We made sure that y+ $<$ 1 for the first cell and y+ $<$ 5 close to the walls. Meshing is an iterative process. During the simulation, if it is observed that convergence is not smooth or is diverging, then this hints that the meshing may need some refinement. The mesh we have generated uses nearly 1.3 million nodes and 7.5 million elements.

The turbulence was modelled with the k-ϵ, k-ω, Shear Stress Transport (SST), BSL Reynolds Stress (BSL) and SSG Reynolds Stress (SSG) models. The k-ϵ, k-ω, and SST models are two equation eddy viscosity models. In the k-ϵ model, transport equations for the turbulence kinetic energy and turbulence dissipation are solved and the turbulent viscosity is determined from these quantities. The SST model applies a blending function that activates the k-ω model near the wall and k-ϵ model in the outer region. The SSG and BSL models are second-order closure models in which transport equations for the individual Reynolds stresses are solved. The SSG model is closed with an ϵ equation while in the case of the BSL model there is a blending between ϵ and ω equations similarly to the SST model.[3]

We now move on to the setup of the simulation. The setup involves specifying all necessary boundary conditions (BCs) and initial conditions of the problem. Each element in the newly generated mesh must be told exactly how to interact with the fluid. In our problem some of the more important things we specified were:

- The fluid filling the fluid domain - water
- The faces of the submarine being no-slip walls
- The inlets being the front face and one side face of the box
- The Cartesian components of the velocity of the water at the inlet - speed of 6.5 knots at the desired drift angle
- The turbulence model used for the simulation
- Residual target of 1e-6 for convergence with a maximum of 10 coefficient loops

The full list of quantities to specify is too numerous to list everything, but this small sub-list should give you a good idea as to what must be specified for these simulations to function correctly.

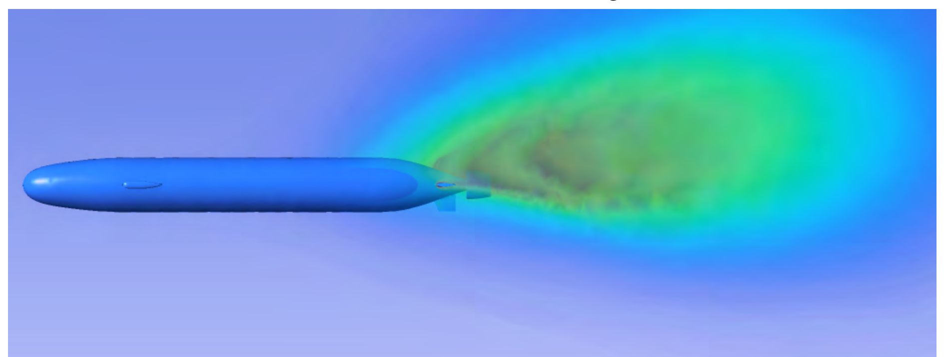
Once that's completed, we can begin performing the simulation. Since we have access to the IRIS HPC cluster in Luxembourg, we can complete such a task much faster with parallelisation – which is quite the upgrade from performing the simulation in serial on your local PC. Since most of the heavy lifting has been done in the setup process, at this point we can rely on the cluster to crunch the numbers once we submit. The program we used for these simulations was ANSYS, meaning we simply need a small script to specify the number of nodes and cores to use and ANSYS takes care of the rest. We ran our simulations on two nodes with 28 cores each as this allowed for fast simulations without having to wait too long for sbatch to allocate us the cores.

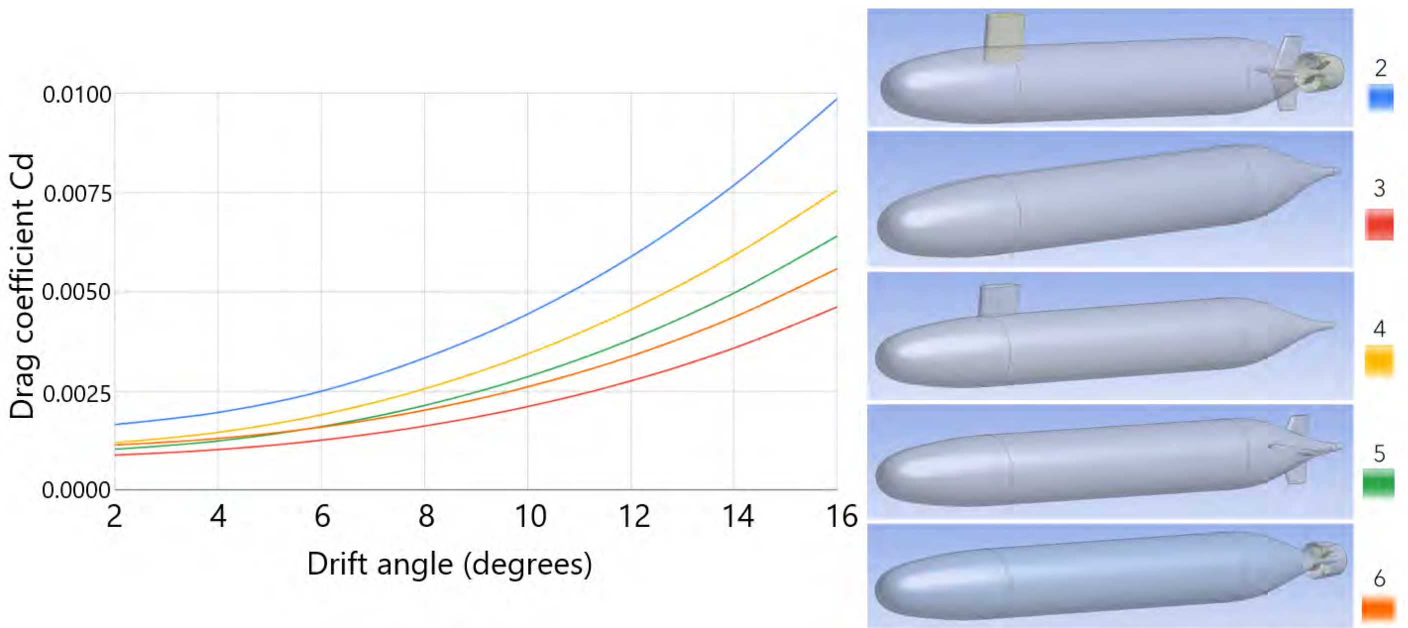
Finally, we arrive at the most exciting part of the process. At this point we see if the conditions we set prior have correctly shown the physical properties we were after – or if we're heading back to the drawing board. Once the files have transferred from the HPC cluster to our local PCs the post-processing begins!

Within post-processing, you are able to select the physical quantities calculated (such as fluid velocity, pressure, temperature etc.) and visualise them on the geometry itself. This allows for a very useful sanity check, as we can see if the simulation has given us physical behaviour, plus we can look at some very cool looking pictures and videos!

Results

We found that the $k - \epsilon$ model was the only turbulence model to greatly differ from all the others. This can be seen in the results on the next page, where the $k - \epsilon$ model predicts the drag coefficient to be higher than the other models for all drift angles.





0.0125
0.0100
0.0075
0.0050
0.0025
0.0000
Drag coefficient Cd
0 2 4 6 8 10 12 14 16
Drift angle (degrees)
k-epsilon
SST
k-omega
BSL
SSG

Figure 2: Drag coefficient against drift angle for different turbulence models. Note that the orange line (SSG) overlaps the lines of SST, k-ω, and BSL.

We also found that configurations with more parts attached to the submarine model, and increasing the drift angle increased the value of the drag coefficient recorded in general.

Discussion & Conclusion

We believe the $k - \epsilon$ model gave different results due to its poor handling of no-slip walls. For this reason, we would believe that the values given by the other models are more accurate. In conclusion, we simulated different configurations of the DARPA Suboff submarine travelling through water at different drift angles at 6.5 knots. Our results showed that configurations containing more parts had a larger drag coefficient, drag coefficient increased with drift angle, and we have reason to believe the $k - \epsilon$ model is a poor model for simulations such as this. In future, work on this topic could include comparison to experimental results.

References

[1] DTIC ADA227715: Investigation of the Stability and Control Characteristics of Several Configurations of the DARPA Suboff Model (DTRC Model 5470) from Captive-Model Experiments

[2] DTIC ADA210642: Geometric Characteristics of DARPA (Defense Advanced Research Projects Agency) SUBOFF Models (DTRC Model Numbers 5470 and 5471)

[3] ANSYS Inc,ANSYS CFX-Solver Modeling Guide, ANSYS CFX-Solver Theory Guide, 2007.

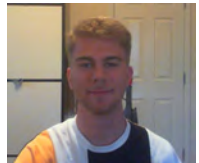
Acknowledgements

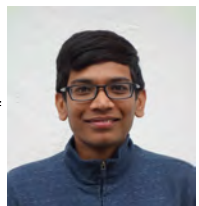
We would like to extend our thanks to the team working at PRACE on the Summer of HPC for allowing us this opportunity to expand our knowledge of HPC systems this summer, especially considering the adverse circumstances that they had to overcome to ensure it still went ahead. We would also like to thank the University of Luxembourg coordinators and our mentor Ezhilmathi Krishnasamy for the guidance throughout our project.

PRACE SoHPCProject Title
Submarine Computational Fluid Dynamics

PRACE SoHPCSite
University of Luxembourg, Luxembourg

PRACE SoHPCAuthors
Matthew Asker, The University of Manchester, United Kingdom
Shiva Dinesh, Friedrich-Alexander University, Germany


Matthew Asker


Shiva Dinesh

PRACE SoHPCMentor
Ezhilmathi Krishnasamy, University of Luxembourg, Luxembourg

PRACE SoHPCContact
Matthew Asker, The University of Manchester
Phone: +44 7730 044781
E-mail: matthew.asker@gmail.com
Shiva Dinesh, Friedrich-Alexander University
Phone: +49 1522 3280 943
E-mail: shiva.d.chamarthy@fau.de

PRACE SoHPCSoftware applied
ANSYS, Salome

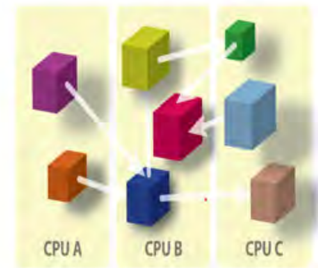
PRACE SoHPCMore Information
www.ansys.com

PRACE SoHPCAcknowledgement
Project co-mentor: Dr. Sebastien Varrette
Site co-ordinator: Prof. Pascal Bouvry

PRACE SoHPCProject ID
2021

Novel HPC Models

Rafał Felczyński, Ömer Bora Zeybek



Aim of the project is making a comparison between novel parallel programming models. Several methods are used for benchmarking.

HPC systems get more heterogeneous in nature. Therefore, it is sometimes feasible to have a single programming model take advantage of it. Novel programming models, for example, AdaptiveMPI, Charm++, XcalableMP, Thrust and Kokkos can take an advantage of the computation in heterogeneous architecture platform. It is therefore to analyze their limitations and make a comparison between them on a HPC architecture.

Iris supercomputer of the University of Luxembourg, which is a heterogeneous system, is used throughout the project. Novel programming models are tested on Basic Linear Algebra Subroutines(BLAS)[1] as well as some real case applications related to biomedical engineering.

Kokkos

Written in C++, Kokkos aims for producing performance portable applications. It supports CUDA, HPX, OpenMP and Pthreads as backend programming models.

XcalableMP

XMP is a C and Fortran extension. It is directive-based and also provides a MPI interface.

Thrust

Thrust is written in C++ and it resembles the standard template library. It supports CUDA and OpenMP backend.

Charm++

Charm++ has unique features, like dynamic load balancing. It is based on C++, and it is object-oriented.

AdaptiveMPI

AdaptiveMPI is built on Charm++. It is an implementation of MPI.

Results

To give general information about BLAS routines, they provide basic vector and matrix operations. These are the routines we used for our tests:

- Ddot (Dot product)
- Dgemv (Matrix * vector)
- Dgemm (Matrix * matrix)
- Dtrsm (Solve matrix equation)

All of them use double precision variables. First routine performs dot product, second routine performs matrix-vector multiplication, third routine performs matrix-matrix multiplication, and the last routine solves a triangular matrix equation with multiple right-hand sides.

Here is an example comparison for Kokkos and Thrust:

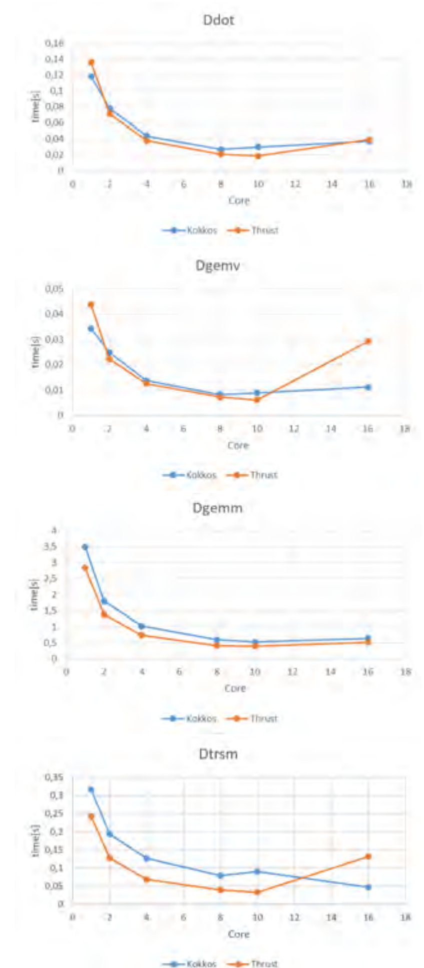


Figure 1: Timings for Kokkos and Thrust

Kokkos and Thrust seem close to each other. While Thrust seems a little better

overall, its performance decreases when number of cores passes a certain value. This may be due to synchronization between the threads.

XcalableMP and Charm++ were also tested with some BLAS routines. The same four routines were picked for the tests as previously and these models were tested for some different matrix and vector sizes and for different number of cores.

Some of the results obtained are presented in form of tables and plots.

cores	vec1 length/vec2 length 1k/1k	1M/1M	10M/10M	100M/100M
	Timings [s]			
1	0,000121	0,00358	0,04103	0,413035
2	0,000137	0,005875	0,064759	0,642481
4	0,000095	0,007267	0,073187	0,721455
8	0,00006	0,007184	0,076824	0,764299
10	0,00013	0,007244	0,077559	0,781315
16	0,000132	0,007322	0,078603	0,780482

Table 1: Timings for XMP dot product operation

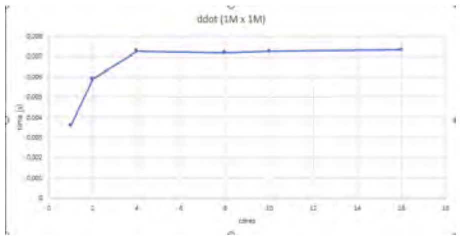


Figure 2: Timings for XMP dot product operation

One can see that the dot product calculation of two vectors done sequentially is much faster than done in parallel. It seems to work as expected because dot product is a pretty simple operation of just going through vectors but the overhead of copying the data back and forth (which is required by this framework) is bigger than just going though the data sequentially. For example going through one Million of elements is in Big O notation equal to O(1M) but execution in for example 2 cores requires O(0.5M) for cores + 2*O(0.5M) for copy half of the two vectors. The copying can be optimised a little be the framework but it still is a huge part of execution time.

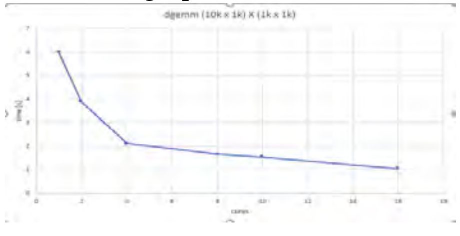


Figure 3: Timings for XMP matrix-matrix multiplication

For matrix-matrix multiplication situation is much better because copying the data is done in linear time O(R*C1 + C1*C2) and the operation of multiplication is done in O(R*C1*C2) so there is a huge difference when multiplication is done in parallel and copying time does not matter that much.

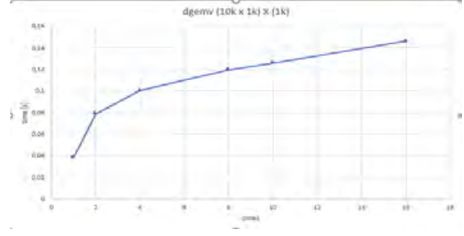


Figure 4: Timings for XMP matrix-vector multiplication

For matrix - vector multiplication situation is similar to dot product calculation and for triangular matrix solving is similar to matrix-matrix multiplication.

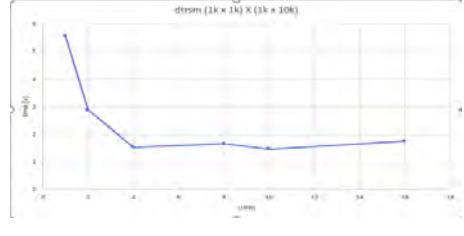


Figure 5: Timings for XMP dtrsm operation

One can see that when the number of cores exceed some value, the time gain does not increase anymore because of context switching and synchronisation that takes place.

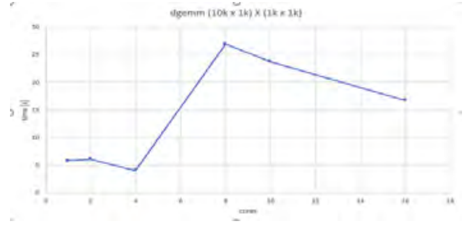


Figure 6: Timings for Charm++ matrix-matrix multiplication

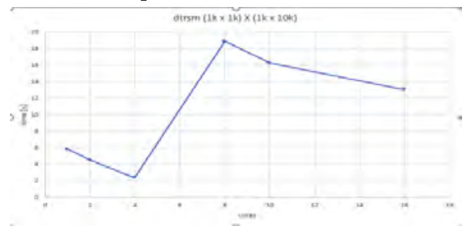


Figure 7: Timings for Charm++ dtrsm operation

For Charm++ the same blas routines were used but with this framework there were some issues. Up to certain number of cores the execution was similar to the one with XcalableMP but when the number of cores exceeded this thereshold, the time suddenly soared. For example for vector of 100 millions of elements, the time rises from about 2 seconds to 130 seconds. This is probably caused by memory allocation synchronisation and network polling that this framework is doing even though the number of available cores is much higher than the used ones and program was executed locally on one node. What is interesting that this not always takes place. Sometimes it works normally and sometimes does not. There are some bugs reported on the framework's Github page so it may be fixed in the future.

In the end 2 biomedical applications were implemented:

1) DNA sequences comparison and creation of a dotplot[2] from them.

2) DNA sequences global alignment. As previously, they were tested with some different DNA sequence lengths and for different number of cores.

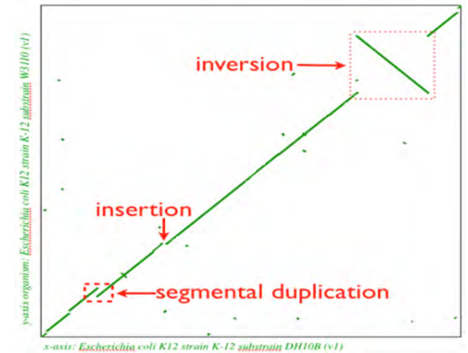
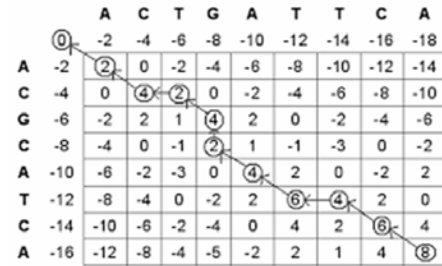


Figure 8: Example of dotplot results

The first application creates a matrix dot plot from two sequences of DNA. Sequence S1 is written in the first column and sequence S2 is written in the first row. For every nucleotide from S1 one has to check every nucleotide from S2 whether they are the same or not. If they are the same, one should put dot in matrix in the related place. If nucleotides are different, the place should be left empty. So the program is simple element-wise checking of two vectors but it allows us to see for example if there are some similarities between organisms. If we compare two related sequences, we can see from the generated picture that there are some insertions, deletions, duplications and so on.

The second program takes two DNA sequences and aligns them together, connecting related nucleotides and making gaps in sequences if there may be some insertions and deletions. The Needlman-Wunsch algorithm[3] was used in this program. First one has to create a scoring matrix of size (length(S1)+2) x (length(S2)+2). Sequence S1 is written in the first column, starting from the third row. Sequence S2 is written in the first row starting from the third column. Then zeros should be places into ma-

trix top-left 2x2 square. One has to assume 3 constant scores – for match, mismatch and gap. The algorithm uses dynamic programming. It means that the next value depends on the previous one.



		A	C	T	G	A	T	T	C	A
	0	-2	-4	-6	-8	-10	-12	-14	-16	-18
A	-2	2	0	-2	-4	-6	-8	-10	-12	-14
C	-4	0	4	2	0	-2	-4	-6	-8	-10
G	-6	-2	2	1	4	2	0	-2	-4	-6
C	-8	-4	0	-1	2	1	-1	-3	0	-2
A	-10	-6	-2	-3	0	4	2	0	-2	2
T	-12	-8	-4	0	-2	2	6	4	2	0
C	-14	-10	-6	-2	-4	0	4	2	6	4
A	-16	-12	-8	-4	-5	-2	2	1	4	8

Figure 9: Example of DNA alignment scoring matrix

Fill the second row and the second column with scores obtained by subtracting the value of a gap from the previous score. In the example the score assumed for gap equals 2. When going through the scoring matrix element by element from top-left corner and calculating next score one should use „match" score if currently related nucleotides are the same and „mismatch" if they are not. To explain how it works lets walk through the example. Starting from the third row and the third column, we calculate current score by taking the maximum from 3 values: the one above minus gap, the one on the left minus gap and the one on diagonal plus match or mismatch, depending on whether the related nucleotides are the same or not. If the whole matrix is filled with scores we go bottom-up from the bottom-right corner of the matrix and we look for a way to the top, checking which cell the current score originates from. It seems like reversing the previous algorithm. If the path goes through the diagonal, we write down nucleotides. If it goes to the left or to the top, we write down a dash in sequences.[4]

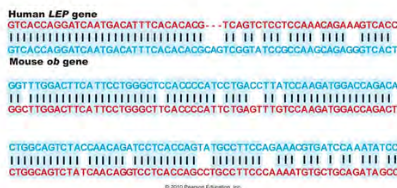


Figure 10: Example of DNA alignment result

Finally we get the result that can be seen in the above picture. Similar algorithm can also be used for multiple DNA sequences alignment but it gets more and more complicated.

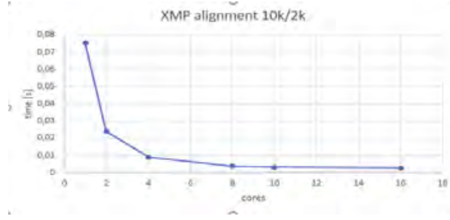


Figure 11: Timings for XMP DNA alignment program

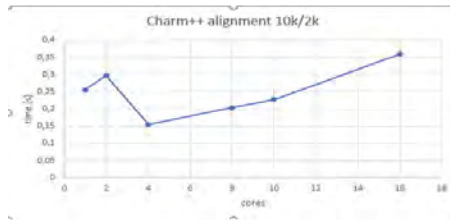


Figure 12: Timings for Charm++ DNA alignment program

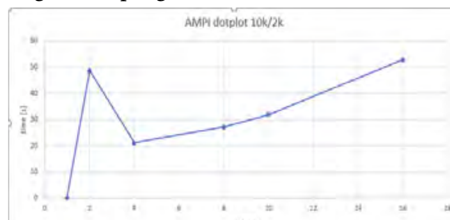


Figure 13: Timings for AMPI dotplot program

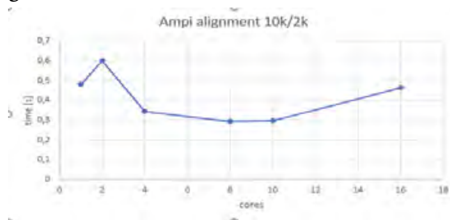


Figure 14: Timings for AMPI DNA alignment program

Just by looking at the timings for XMP, one can see many similarities to the previous plots. When using XMP for parallel DNA alignment one can get huge drop in time spent for calculations. The results are very good. One can imagine aligning thousands of very long sequences every day and how much time can be gained by parallel execution of the program. For charm++ the results also are as previously – some weird framework behaviour just messes the results up. But for up to 4 cores there is some time gain, not that much as it was for XMP though. These biomedical applications have also been tested with AMPI, which is built on top of charm++. The results obtained from that framework also are not good. The reason may be the charm++ environment mentioned before. This is especially interesting if you look at the sudden time increase from 0.07s to 48s which is caused by only increasing the number of cores from 1 to 2.

Conclusions

Although the documentation is quite poor which make programs not so easy to write, XcalableMP seems to be the most stable framework and have quite predictable behaviour. Additional thing to consider is the amount of memory used by this framework due to its lack of memory run-time allocation and the possibility to exchange the data between the other processes.

Charm++ and AMPI are really nice to write programs in because they are intuitive and pretty well documented but not all the documented features really work. These frameworks use network to pass messages and share data and it may be really tricky sometimes and cause a lot of problems.

Not all the programs are good to parallelisation. Many things have to be considered and sometimes, especially if the size of the problem is not so large, because the overhead of creating the processes, context switching, data copying etc. can be meaningful.

References

[1] BLAS routines

[2] Dot plot algorithm

[3] Needleman Wunsch algorithm

[4] Chakrabarti, S. D. (2011). DNA Sequence Alignment by Parallel Dynamic Programming.

PRACE SoHPCProject Title
Novel HPC Parallel Programming Models for Computing (both in CPU and GPU)

PRACE SoHPCSite
University of Luxembourg, Luxembourg

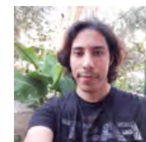
PRACE SoHPCAuthors
Rafał Felczyński, Wroclaw University of Science and Technology, Poland
Ömer Bora Zeybek, Boğaziçi University, Turkey

PRACE SoHPCMentor
Dr. Sebastien Varrette, ULux, Luxembourg
Dr. Ezhilmathi Krishnasamy, ULux, Luxembourg

PRACE SoHPCAcknowledgement
Project Mentor: Dr. Sebastien Varrette
Project Co-mentor: Dr. Ezhilmathi Krishnasamy
Site Co-ordinator: Prof. Pascal Bouvry

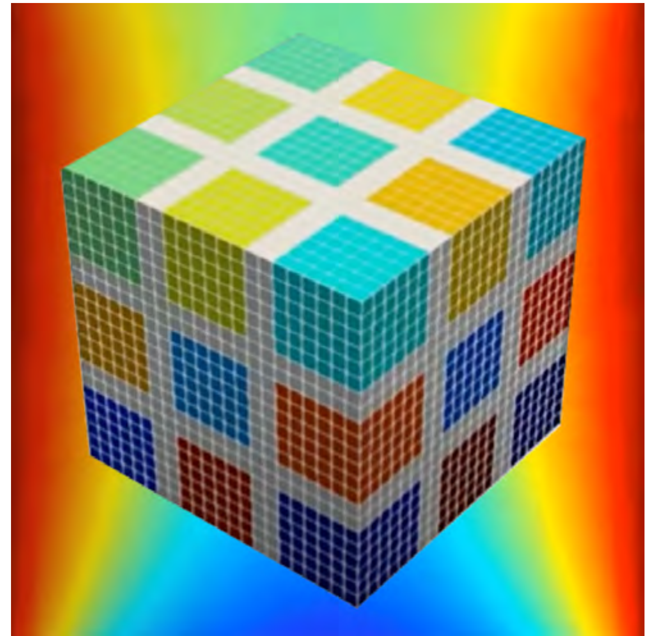
PRACE SoHPCProject ID
2022

Rafał Felczyński



Ömer Bora Zeybek

Get ready for exascale computing – exploring different ways to combine and exploit both distributed and shared memory architectures



Hybrid Programming with MPI+X

Sanath Keshav,
Kevin Mato,
Clément Richefort,
Federico Sossai

What happens when your dataset is so big that it does not fit on your computer? Let us have a look at how to distribute both work and data on a supercomputer and analyse the main aspects of hybrid programming.

Moore's law is still a real thing but processors have physical limits anyway: it is not possible to take the number of core units in the same chip to the extreme and this is why nowadays supercomputers are composed of multiple nodes that cannot share the same memory. Here *hybrid programming* comes into play.

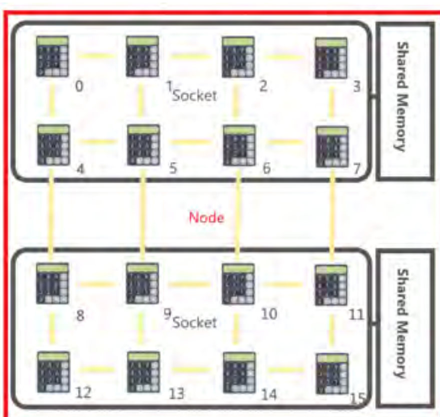


Figure 1: Schematic view of an VSC3 node with its two sockets and their cores.

What does *hybrid* mean?

Modern supercomputers are composed of lots of *computing nodes*, each one equipped with several CPUs (also called *sockets*) that by itself consist of multiple *cores*, as represented in Figure 1. All cores within one node can directly access the same memory, that's why we say that this memory is *shared*. On the other hand, cores on different nodes cannot access each other's memory without an explicit communication, that's why this is called *distributed* memory. Whenever a program exploits parallelism for both distributed and shared memory architectures we talk about *hybrid programming*.

In this work we made extensive use of the *Message Passing Interface* (MPI), the *de facto* standard to orchestrate distributed computing. Since several hybrid programming options are explored in this work, we will use the expression *pure MPI* indicating any implementation oblivious to the shared memory.

Hybrid programming, as it is usually referred to, may combine two different standards like MPI and OpenMP (a very popular way of work sharing within shared-memory regions) to take advantage of the underlying architecture. However, in this project we explored also other paths, for instance, by applying MPI's own built-in shared-memory model that allows to exploit shared and distributed memory at the same time by means of the so-called *one-sided shared-memory* features. We also made use of a combination of MPI plus OpenACC to discuss the pros and cons of exploiting accelerators such as GPUs.

Solving a differential equation

As a prototypical code, we focus on the numerical solution of the Helmholtz equation, which is shown here in 2 dimensions (2D)

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} - \alpha u = f$$

by using the finite difference method. When the datasets involved get huge and infeasible to solve on a single computer, the natural solution is to divide and conquer. The physical domain of

the problem is cut into smaller domains and numerically solved on multiple cores.

How to cut the domain?

The domain can be cut only in a particular direction to obtain stripes (in 2D) and slabs (in 3D). Although we have balanced load on every process, the number of interface points is very high and this does not scale well. A more elegant solution is to cut the domain in all directions using virtual Cartesian topologies (see the top of Figure 2 for 2D and the title figure for 3D). This way we have a balanced load as well as minimal interface points for communication. This approach scales well with very large numbers of processes.

Jacobi in a nutshell

To solve the equation we choose the Jacobi algorithm. In layman's terms the Jacobi iteratively computes a grid of points from a previous one in which every point depends only on its neighbours. Thinking about this for a second, you'll realise that if we split this grid in subgrids (in order to feed different distributed nodes), points at the boundaries will depend on points which are not in the same subgrid. A data communication among subgrids is therefore needed.

Halo communication and stencil

The local domain is discretised and so-called *halos* are added around the local datasets of each process. These halos will hold a copy of the borderline data communicated from their neighbours.

All the processes send and receive their borderline data in all spatial directions, as can be seen in Figure 2. For an efficient halo exchange we use MPI nonblocking communication allowing for an overlap of communication with other communication operations, thus keeping the communication subsystem as busy as possible.

Once this halo exchange is completed, the Jacobi stencil computes the next iteration using the solution of the previous iteration. From the stencils indicated in the lower part of Figure 2 it becomes clear that the halo data is needed to be able to do the Jacobi update in the outer parts of the local datasets.

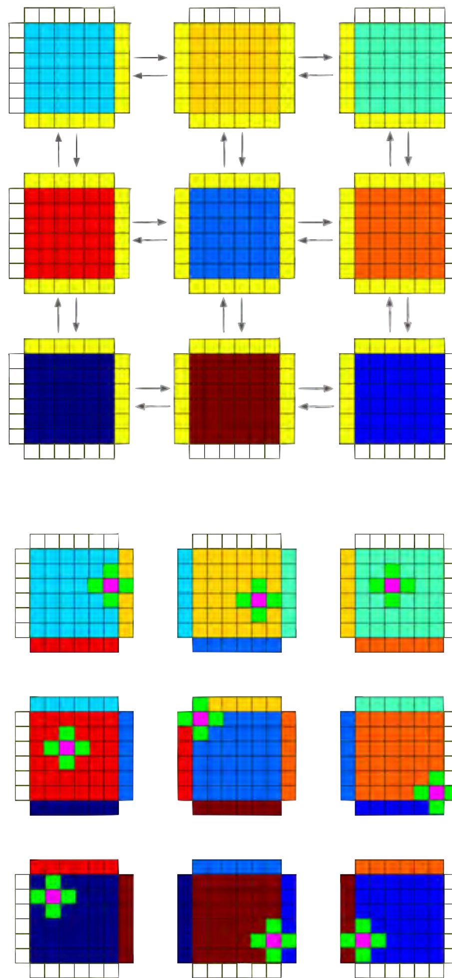


Figure 2: 2D domain decomposition before (top) and after (bottom) halo communication with the stencil at various locations.

Combining MPI and OpenMP

While pure MPI is designed for distributed memory (and therefore even does halo communication within a shared-memory area), OpenMP is designed for shared memory and can be exploited within a single node that contains a physical shared memory. Fundamentally, each MPI process can have multiple OpenMP threads executing simultaneously. Several configurations can be opted for including involving one MPI process per node or one MPI process per socket while the remaining cores will be used by OpenMP threads. The efficiency of the configuration is influenced by the hardware architecture. Combining MPI and OpenMP reduces the need for replicated data within a shared-memory area and can have interesting designs that can overlap computation and communication (but this latter option has not yet been exploited during this project).

Different options to combine MPI and MPI one-sided shared memory

The principle of employing MPI to do the node-to-node communication of halo data and using MPI's one-sided shared memory features within the shared memory of the individual nodes or sockets is very similar to MPI+OpenMP but offers more freedom.

Shared memory within each socket

In this configuration a shared-memory area is allocated on each socket that can be directly accessed by all processes on this socket. The program implementation of this option is straightforward.

Shared memory within each node

Another possibility is to allocate a shared-memory area within an entire node. There is quite a difference in performance whether this shared memory is contiguous or not. With contiguous memory the program will be much easier to write as it is sufficient that only one process of a node allocates the shared memory. The processes that run on the same socket will have a fast access to this contiguous shared-memory area, but for the processes running on the other socket the memory access will be much slower (see the left part of Figure 3).

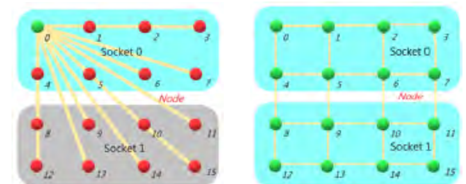


Figure 3: MPI one-sided shared memory allocated as contiguous (left) or as noncontiguous (right) shared-memory area within the node.

This can be avoided, if the shared-memory area is allocated in a non-contiguous way. However, writing the program is more tricky, as each process has to allocate its own portion of the noncontiguous shared-memory area that will be placed in the memory of its own socket allowing for fast access. In order to directly read from the neighbours' data (which substitutes the halo communication inside the shared-memory area) the processes have to

query the memory addresses from their neighbours (see the right part of Figure 3).

Only one process communicates

MPI permits to decide which processes will be communicating with others. Thus, it is sometimes relevant to ask ourselves what is the best between making every process communicating or only a few of them. An idea here is to let only one process of a shared-memory area do the halo exchange with others as depicted in Figure 4. It reduces the number of communications, but the pieces of data to send to or receive from others are obviously bigger. For the 2D version, the option that only one process communicates together with shared memory within each socket gives a pretty interesting compromise between the number of communications and the size of the data to be sent or received.

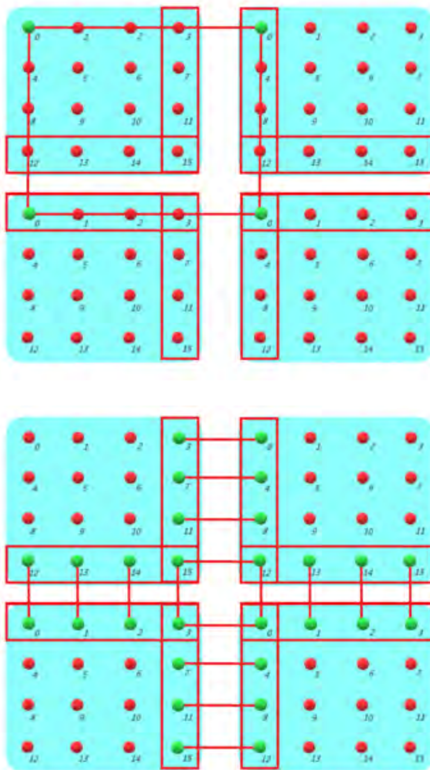


Figure 4: How many processes do the node-to-node halo communication? Only one process per node (top) or all boundary processes (bottom), shown between 4 nodes.

Boundary processes communicate

A different option is to involve all processes that work on the boundaries of the shared-memory dataset in the halo communication, each communicating only a certain fraction of the total halo needed as illustrated in Figure 4.
Two Cartesian grids are necessary so that each process knows if it belongs to a border, and, if so, to whom and from whom it has to send and receive the halo data. One for nodes or sockets, the other for processes within the shared-memory area.

Hybrid on CPU nodes

The importance of pinning

High performance does not only come from an accurate implementation, the so called process pinning plays a huge role as well. MPI processes or OpenMP threads may not have a specific unit on which to run, therefore they can float from core to core, introducing useless overheads. The so-called *pinning* avoids this bad behaviour, allowing us to establish a one-to-one correspondence between processes/threads and cores. Let's see how pinning can affect performance when scaling our Jacobi program with multiple cores of one node of VSC4 (see Figure 5).

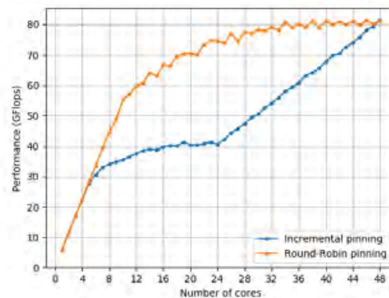


Figure 5: Intra-node performance of pure MPI on a single node of VSC4.

The simplest pinning we can think of is *incremental* pinning: process 0 is assigned to core 0, process 1 to core 1 and so on, that means we fill socket 0 consecutively before starting to use socket 1 at all. Recalling that a node on VSC4 is composed of two sockets, each with 24 cores, when we use incremental the first 24 processes are pinned in the first socket, saturating the bus that connects cores and memory.

We use the expression *round-robin* when a process is pinned to a core on the first socket, the next process to the second socket, the third to the first socket again in an alternating way until all processes are pinned. Doing so, both sockets are filled progressively, and the memory bandwidth is saturated gradually. The reader should notice that no matter what fair pinning is set, when all cores are used performance will eventually reach the same limit.

In the case of hybrid programming with MPI and OpenMP, in addition to correct process pinning, ensuring thread affinity is vital to extract optimal performance. This avoids multiple threads/processes sharing a physical core which would drastically deteriorate performance due to the threads competing for resources. Proper thread pinning is necessary for scaling with more OpenMP threads.

Comparing the results

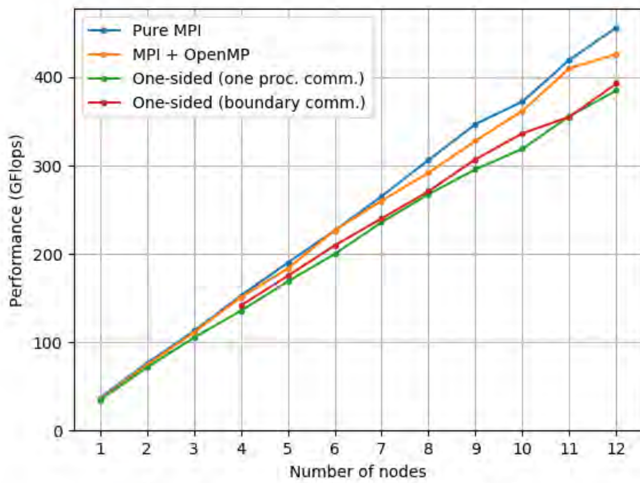
The inter-node strong scaling results presented in Figure 6 refer to the VSC3 cluster. We can see that the pure MPI version dominates the others in both 2D and 3D versions, and conversely the one-sided versions with only one processes per socket involved in the communication appear slower. Among the other hybrid versions, the 2D case is dominated by the MPI + OpenMP version, but when dealing with a 3D problem the one-sided with boundary process communications prevails over the other hybrid versions. Performance depends on the cluster and its features. Even if it is very hard to beat the pure MPI version of the program, hybrid versions can be very interesting, particularly on clusters having a poor communication bandwidth.

Hybrid with multiple GPUs

Since nowadays many clusters are equipped with accelerators such as GPUs we also provide an implementation that allows to use multiple GPUs on the same node or on multiple nodes.

Here the GPUs are tackled with OpenACC (an other option would have been to use CUDA directly). The choice for the pragma-based OpenACC was made for the great trade-off between development time and speed-up. In addition, it offers the possibility to compile the code not only for a target GPU but also in a *multicore* version and in a *serial* or *host* version (in our case pure MPI). The multicore option is offered as a possible replacement for a combination of MPI and OpenMP.

Inter-GPU communication will be done by MPI just via the host CPU



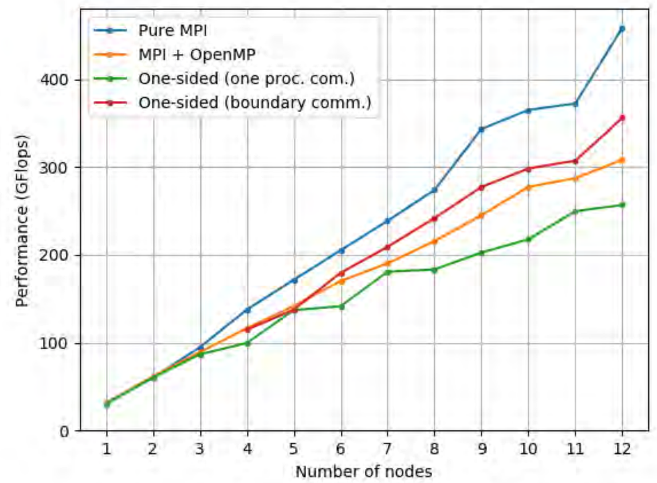


Figure 6: Performance of various implementations of the Jacobi solver in 2D (left) and 3D (right) on VSC3 nodes. MPI + OpenMP was done with one MPI process per socket each engaging eight OpenMP threads. With One-sided the plots show two configurations, only one process communicates per socket with contiguous shared memory and all boundary processes communicate with contiguous shared memory on a socket.

and the PCI bus connecting GPU and CPU while MPI will treat all memory allocated on the GPU device as if allocated on the host CPU via the so-called CUDA-awareness. At startup each MPI process allocates and initialises its own matrices on the GPUs in order to avoid excessive transmission of data through the PCI bus which is the bottleneck. Thanks to asynchronous execution we can hide the communication time between processes that pass halo data coming straight from the devices.

What about the scaling?

Since the memory of a GPU isn't comparable in size to the memory of the host CPUs (node), we focus on studying the weak scaling of our multi GPU implementation. For this study we used one special node of VSC3 with eight GPUs (NVIDIA GeForce GTX 1080 v6.1, with 8.5 GB of memory). In this case the host has two CPUs with four cores each.

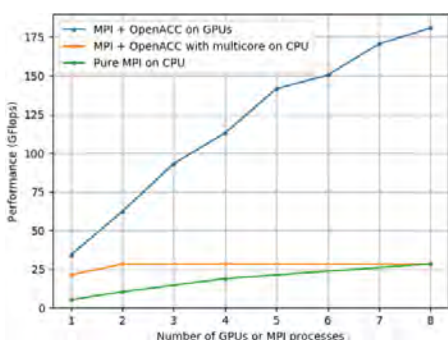


Figure 7: Performances of MPI + OpenACC on GPUs, MPI + OpenACC with multicore on CPU and just pure MPI on the host CPUs.

Is it really better?

Figure 7 shows the weak scaling results of the 2D Jacobi solver using up to eight GPUs, and it is compared to the weak scaling performances of OpenACC's multicore compilation and of the pure MPI version of the code, all of them on the same node. It is easy to see that the usage of accelerators is outperforming the other two compilation options. By using eight GPUs the speed-up achieved is 6.4× as compared to the multicore (engaging all cores of the host CPUs), and 5.2× against just using one accelerator. A total win.

Conclusions

In this project we implemented and analysed several configurations of a Jacobi solver, taking into account the physical properties of the memory as much as possible. While on one hand a pure MPI implementation seems to be the fastest most of times, on the other, the extension to the 3D version enabled us to conclude that when it comes to hybrid programming, memory configurations and other communication aspects can have a noticeable impact on performance.

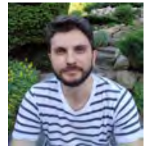
To further improve the performance of the hybrid versions, future work might focus on reducing the synchronisation overhead of the shared-memory programming models as well as avoiding idle times by overlapping communication and computation.

PRACE SoHPCProject Title
Improved performance with hybrid programming

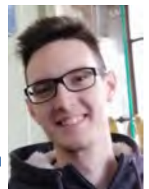
PRACE SoHPCSite
VSC Research Center, TU Wien, Austria

PRACE SoHPCAuthors
Sanath Keshav,
Kevin Mato,
Clément Richefort,
Federico Sossai

Sanath Keshav


Kevin Mato

Clément Richefort


Federico Sossai

PRACE SoHPCMentor
Claudia Blaas-Schenner and Irene Reichl, VSC Research Center, TU Wien, Austria

PRACE SoHPCContact
Sanath, Keshav, Universität Stuttgart
E-mail: sanathkeshav.mysore@gmail.com

Kevin, Mato, Politecnico di Milano
E-mail: kevin.mato@mail.polimi.it

Clément, Richefort, Polytech Lille
E-mail: richefort.clement@protonmail.com

Federico, Sossai, University of Padua
E-mail: federico.sossai@studenti.unipd.it

PRACE SoHPCSoftware applied
MPI, OpenMP, OpenACC, C, C++

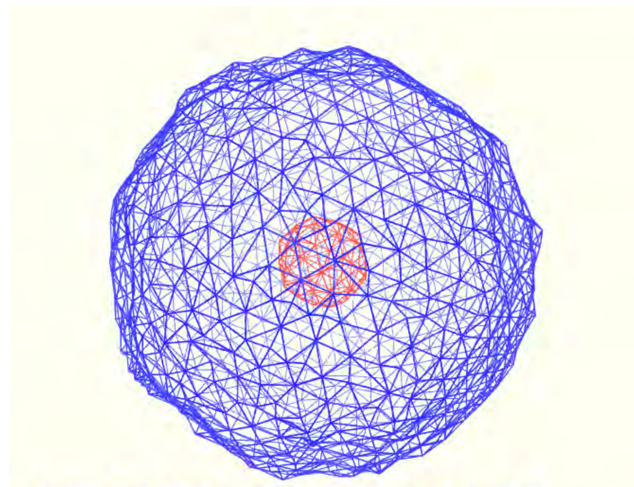
PRACE SoHPCMore Information
MPI-Forum.org
OpenMP.org
OpenACC.org

PRACE SoHPCAcknowledgement
We especially thank our mentors Claudia, Irene and David for their wonderful help and pedagogic explanations, and of course the Vienna Scientific Cluster in general for having given us access to their super supercomputers!

PRACE SoHPCProject ID
2023

Biomolecular Meshes

Aaurushi Jain, Federico Camerota, Jake Love



This summer our team worked on an implementation of the marching tetrahedra algorithm. The program was partially ported to the GPU and was used to create visualisations of the electron density field around complex proteins.

Three dimensional scalar fields are inherently hard to visualise graphically. The two main approaches used are (i) to create an isosurface from the field or (ii) to use a volume rendering technique. This report focuses on the former.

An isosurface can be considered to be the three dimensional equivalent to a contour. In the same way that a contour represents a set of points of equal value in two dimensional space, an isosurface represents a set of connected points of equal value in three dimensional space. By picking an appropriate isovalue, one can create a mesh of points that all share the characteristic of exhibiting identical values of the underlying scalar field. Although this does not give a complete visualisation of the entire system it can highlight important features of the field and by varying the isovalue a good intuition of the field's structure can be formed.

The marching tetrahedra algorithm is a method of extracting a mesh representing a scalar field's isosurface at a specific isovalue. It is the junior version of the seminal marching cube algorithm.[1] These type of algorithms are important for a wide range of applications, a common example being medical imaging. For example, from a dataset of tissue densities (CT scan) a mesh can be computed that represents internal structures of the tissue. This allows abnormalities — such as tumours — to be identified at high levels of confidence.

In this project, however, we used the marching tetrahedra algorithm to extract and visualise the electron density field in proteins. There are three major steps in the pipeline that creates our desired mesh. First, the scalar field is computed using the approximation of Slater densities. Next, an isosurface is extracted from application of the marching tetrahedra algorithm. Finally, a postprocessing step is introduced to reduce the number of elements in the mesh via vertex and edge reduction techniques.

Slater Density Field

To begin with, a volumetric dataset representative of the biomolecule's spatial extent is set up and computed. We decided for the simplest geometry of a regular cubic grid. The coordinates of the atoms in the biomolecule gave rise to a corresponding origin. Within a nested loop over all 3 grid dimensions, each grid point was identified and became subject to electron density calculations. The approximation followed was to make use of the sum of atomic Slater functions[2] centred on each of the atoms in the biomolecule. Care had to be taken of properly converting atomic units whenever distances between atoms and grid points were to be taken into account (Bohr → Å). The approach turned out to be compute-intensive and the corresponding function, map2grid(), was likely to become a target of further optimizations and refinement strategies.

Marching Tetrahedra Algorithm

Tetrahedral Decomposition

The first step in the Marching Tetrahedra algorithm is to decompose the domain into individual tetrahedra. From the requirement of a space-filling decomposition a frequently used approach is to go in two sub-steps, first divide the volume into cubes, then decompose each cube into tetrahedra. In our particular case, the input data consists of a 3D grid (see section "Slater Density Field") where individual grid points should also become the vertices of our cubes and tetrahedra. At first, we used a cube decomposition into six tetrahedra. The advantage of this approach is its simplicity since each cubic element can be processed in an identical way. However, the resulting mesh turned out to contain a lot of very small sized triangles.

To improve on this, we switched to a 5-fold decomposition of cubic elements into individual tetrahedra. This is technically more involved because we need to consider two different symmetries in an alternating fashion for neighbouring cubes.[3] In so doing, we managed to remove some of the undesired triangles in the mesh. However, we also quickly realised that there was room for

further improvement. So far, we had been working with "vertex-based representations", meaning that whenever we referred to an object we did so by implicitly referring to its constituting vertices. Instead, one can also make use of edges to identify objects. The key idea here is a method to assign unique identifiers (numbers 1 … N) to edges using only their associated vertices. At first sight it might not seem clear how this could be an improvement. The advantage of an "edge-based representation" arises from the fact that all vertices of the output mesh will also lie on them. Consequently, using such unique edge identifiers for the set of vertices forming the output mesh will result in the latter list to also become unique. This is an important requirement of many additional processing steps lined up in the algorithm. With such a key-enabling feature put in place we could even think of parallelizing the two main tasks in the overall algorithm, i.e. (i) finding intersection points on the edges of individual tetrahedra, (ii) extracting faces to form the actual isosurface. This is because the latter task does not require explicit knowledge of the exact location of intersection points, which can be computed in parallel by the former task. Therefore, extracting faces will actually mean providing triplets of unique edge identifiers, which stresses again the significance of an edge-based representation.

Face Extraction

Each tetrahedron is examined with respect to which of the vertices are enclosed by the isosurface and which are not. Vertices with field value greater than the isovalue are enclosed while vertices with values less than the isovalue are located outside the volume enclosed by the isosurface. If all the vertices of a particular tetrahedron are either enclosed or unenclosed, then we are dealing with a trivial case and can move on to the next tetrahedron. However, if some of the vertices turn out to be enclosed while others are not, then the underlying tetrahedron will define a fragmental part of the isosurface and needs to be further considered for in-depth processing. In such cases a surface element will be extracted following the intersection logic inherent in the marching tetrahedra algorithm.[3] Once all the surface elements have been identified and extracted, the final mesh is established.

To extract a particular surface element, one of three cases can apply. Either one, two or three vertices of the tetrahedron may be enclosed. In the first case, a single triangular surface element will be extracted. This triangle will separate the single enclosed vertex from the other three unenclosed vertices. Similarly, the third case will also result in a single surface element to be extracted, separating the single unenclosed vertex from the other three enclosed vertices. In the remaining case of two vertices being enclosed while the two others are found unenclosed, a quad element (consisting of two adjacent triangles) will be extracted again separating enclosed from unenclosed vertices.

Coordinates of the extracted surface elements — i.e. vertices of individual triangles — are computed via linear interpolation along tetrahedral edges to the desired isovalue. Care is taken to ensure that normal vectors to all surface elements are consistent across the final mesh. Graphical inspections were done with VMD.[4]

Mesh Reduction

The aim of the mesh reduction is to simplify the initial isosurface by removing small and very elongated triangles. This results in a more homogeneous mesh and reduces the number of triangles to be taken into account. We do this because in the associated biomolecular simulation less triangles imply a reduced number of equations to solve. Moreover, the inclusion of many small sized or elongated elements would rather deteriorate than improve the numerical accuracy of the results. In addition to that, reducing the number of surface elements will also ease the amount of required memory and speed up surface processing in general. The method itself is quite simple: (i) identify edges that can be removed, (ii) contract them into a single point, (iii) remove triangles involving that edge. Despite its simplicity, this procedure allowed us to bring down the number of triangles in our test surface from 70k to 10k maintaining the mesh intact and preserving most of its properties. Fig. 1 represents a comparison between initial, unprocessed and reduced mesh structure based on a selected region of our test system.

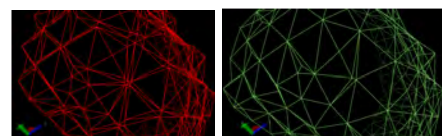


Figure 1: Mesh before (left) and after reduction (right)

Basic Properties of the Mesh

Among the most important properties of a mesh to be considered well-structured are, (i) uniform orientation of vertices in all surface elements (triangles) being arranged in either clockwise or counter-clockwise fashion, (ii) closure of the surface. By differently colouring the first, second and third vertex of individual triangles in red, blue and green, we can graphically check the rotational sense of the vertices in individual triangles. Fig. 2 represents the outcome of such an analysis carried out on one of our early implementations. As can be seen, the arrangement of triangular vertices is not uniform, but randomly switches between clockwise and counter-clockwise. From this it follows that an additional loop over all surface elements is to be run in the end of the program and corresponding normal vectors need to be re-calculated from scratch via application of the cross-product.

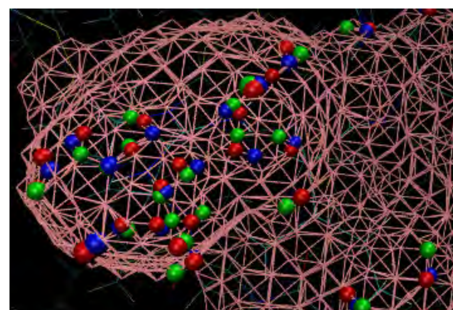


Figure 2: Rotational orientation of vertices of the surface elements (triangles) forming the mesh (of an early version of the program). Going from the first vertex (red) to the second (blue) to the third (green) allows to discern clockwise from anti-clockwise orientation, which is, however, adopted randomly.

With respect to the second basic property of surface closure we made use of the Euler characteristic,[5] X, which is defined as $X = V - E + F$ where V, E and F are the number of vertices, edges and faces of the mesh respectively. It is important to realise that all parameters in X are to be seen as unique properties, e.g. numbers of unique edges, numbers of unique vertices etc.

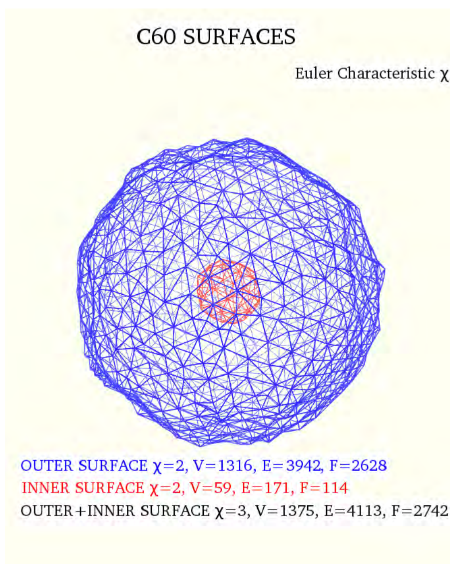


Figure 3: C60 Surfaces

In order to establish confidence in the Euler characteristic, an analysis program was developed and applied to the well-known case of Buckminster Fullerene,[6] C60. The latter exhibits a cage-like ring structure that resembles a soccer ball. Its mesh was expected to be a sphere with $X = 2$. On applying our algorithm we found that C60 gives rise to actually two surfaces, an inner and an outer one (see blue and red regions in Fig. 3). Both inner and outer surfaces were separated and analysed independently. Corresponding Euler characteristics were found to be $X = 2$ for either surface (see Fig. 3 for details of V, E and F). Moreover, when considering the entire list of triangles as compound mesh, X turned out to be 3 (results given in black in Fig. 3). From this it becomes clear that X can be regarded a convenient means to determine whether a mesh describes a single surface of closed shape.

Statistical Analysis of the Mesh

Throughout the development cycle it was helpful to analyse and compare different versions of the code in terms of statistical descriptions of the components forming the mesh. For example, in Fig. 4 three different approaches of domain decomposition are analysed with respect to the size distribution of the resulting surface elements. It immediately becomes clear that the edge-based decomposition (green curve) leads to an increased average size of individual surface elements (0.15 Å^2) and also distributes them over a much larger range than what was observed in the plain decomposition into 5 or 6 tetrahedra (blue and red curves). In addition, the abrupt truncation at the 0 Å^2 point of the latter 2 approaches hint at a large fraction of very small sized triangles yielding large numbers of numerical zeros at the level of accuracy inherent in Fig. 4.

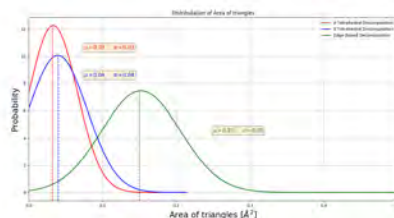


Figure 4: Distribution of triangle area

In a related analysis the length of edges of individual surface elements (triangles) was evaluated statistically and compared between different approaches. The result is graphically shown in Fig. 5. Again, while both 5- and 6-fold tetrahedral decompositions (blue and red curves) show rather linear distributions of steadily increasing edge lengths, the edge-based approach (green curve) clearly has removed all small sized elements and only starts at a certain threshold edge length of approximately 0.25 Å. Moreover, edges longer than 0.40 Å seem to be evenly distributed in the edge-based approach (green) whereas they are increasingly occurring with increasing edge length in the 5- and 6-fold tetrahedral decomposition (blue and red curves).

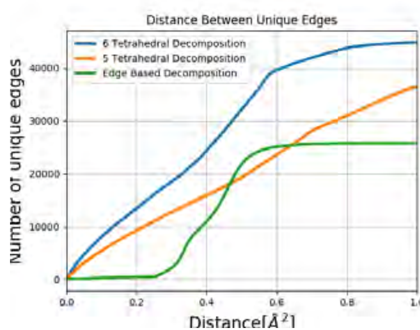


Figure 5: Edge length distribution

Profiling and Early GPU Port

The algorithm's execution time was analysed for relative content of individual functions using profiling tools from ARM/Forge on VSC-3. Initially the total execution time was 44.8 seconds. Profiling revealed a major fraction of the execution time spent in function "slater_density()" which was anticipated as it was repeatedly called for each of the atoms exerting their partial contribution to the density at a particular grid point. After optimising loops and related routines the execution time could be reduced to 18 seconds. Since after repeated profiling of this improved version another function, "map2grid()", was projected out, which had the above function "slater_density()" again called in its innermost loop, it seemed tempting to try a GPU port of function "map2grid()". Plain CUDA, version 9.1.85, in combination with ANSI C was used on an NVIDIA GPU of the Pascal architecture of type GeForce GTX 1080. In so doing the execution time could be further decreased to 1.7 seconds neglecting any higher level optimisations (i.e. memory access patterns) at this point in the development cycle.

References

[1] Lorensen, W. E, Cline, H. E. (1987). Marching cubes: A high resolution 3D surface construction algorithm, ACM SIGGRAPH Computer Graphics 21(4), pp 163–169.

[2] Slater, J. C. (1930). Atomic shielding constants, Phys Rev 36,57-64.

[3] Doi, A., Koide, A. (1991). An Efficient Method of Triangulating Equi-Valued Surfaces by Using Tetrahedral Cells, IEICE Trans E74(1), 216-224.

[4] https://www.ks.uiuc.edu/Research/vmd

[5] https://en.wikipedia.org/wiki/Euler_characteristic

[6] https://en.wikipedia.org/wiki/Buckminsterfullerene

PRACE SoHPCProject Title
Marching Tetrahedrons on the GPU

PRACE SoHPCSite
VSC Research Center, Austria

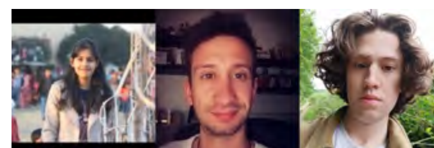
PRACE SoHPCAuthors
Aaurushi Jain, Federico Camerota, Jake Love

PRACE SoHPCMentor
Siegfried Hoefinger, Vienna, Austria

PRACE SoHPCCo-Mentors
Markus Hickel, Balazs Lengyel, Vienna, Austria

PRACE SoHPCAcknowledgement
We would like to express our gratitude to our mentor Siegfried Hoefinger for providing us the guidance throughout the project. Furthermore, We would also like to thank PRACE for giving this opportunity to work at VSC Research centre, Vienna.

PRACE SoHPCProject ID
2024



Aaurushi Jain, Federico Camerota, Jake Love

www.summerofhpc.prace-ri.eu