PROJECT PROPOSAL PHPC-2019

A High Performance Implementation of a solver to the N-Body problem: Barnes-Hut approximation at scale

Principal investigator (PI)	Antoine Weber
Institution	EPFL-SCITAS
Address	Station 1, CH-1015 LAUSANNE
Involved researchers	Only PI
Date of submission	June 9, 2019
Expected end of project	June 20, 2019
Target machine	DENEB GPU Nodes
Proposed acronym	NBODY

Abstract

The N-Body problem is a well-known problem in physics. It was first developed to understand the motion of the sun and planets orbiting around it. It comes with a high $\mathcal{O}(n^2)$ complexity which becomes quickly intractable for big systems. To reduce the complexity for solving the N-Body problem, the use of the barnes-hut approximation was studied. It allows a great reduction of complexity to $\mathcal{O}(nlog(n))$ due to its tree-based architecture.
For the sake of this report, the potential of parallelizing the barnes-but approximation using the famous CUDA parallel computing platform developed by Nvidia was studied. The main part of the application is coded in C++ with calls to .cu files containing the CUDA kernels.

1 Scientific Background

Solving the N-Body problem consists of computing the resulting force for each body in a multi-bodies system. The force acting on body i by body j is as follows:

$$F_{ij} = \frac{Gm_i m_j(q_j - q_i)}{||q_j - q_i||}$$

where G is the gravitational constant, m_i and m_j the masses of the two bodies and q_j and q_i their positions.
To compute the resulting force on a single body, a loop through all the other bodies has to be performed. Doing this computation for all bodies results in the known $\mathcal{O}(n^2)$ computational complexity. Solving the NBody problem the latter way is referred as the $bruteforce$ algorithm.
The barnes-hut approximation introduces the implementation of a tree-based solver. In the 2D case, this tree is referred as a *quadtree* (*octree* in 3D but this project was focused on the 2D problem). Starting from a squared initial space containing all the bodies, it is recursively cut in 4 quadrants until each particle remain alone in a single quadrant. Internal nodes of the tree, being nodes having one or more children, contains the center of mass of all underlying internal and external nodes. External nodes are the end leaf of the tree.
When computing the resulting force on a body, a parameter θ is defined as representing the

threshold at which an internal node and all its underlying bodies can be defined far enough to consider them as a single body. This is known to reduce the complexity of solving the N-Body problem to $\mathcal{O}(nlog(n))$. However, due to the trick of considering multiple bodies as a single one, it remains an approximation. For high precision applications, the barnes-hut algorithm should not be used. Finally, fixing the θ parameter to 0 degenerates the approximation to the brute-force algorithm.

2 Implementations

The application is implemented in C++ with CUDA kernels in C. In order to compare our results to different baselines, 3 implementations were created:

- The Brute Force algorithm was implemented with the use of CUDA kernels.
- A sequential version of the barnes-hut approximation was implemented on CPU.
- A parallel version of the barnes-hut approximation with the use of CUDA kernels.

The code has been fully debugged using the `gdb` debugger. The `valgrind` tool has been used to remove all the memory leaks for the sequential codes. Moreover, different tools were used for profiling the code, such as `nvprof` and `vtune`.

2.1 Optimizations

There is a main bottleneck which was not tackled: I/O. Indeed, for the sake of this study, the I/O were compiled only to run in debug mode for observing the bodies displacements. Whenever the code was run for performance, the I/O parts were not compiled which is why this bottleneck was not investigated.

2.2 Brute Force with CUDA

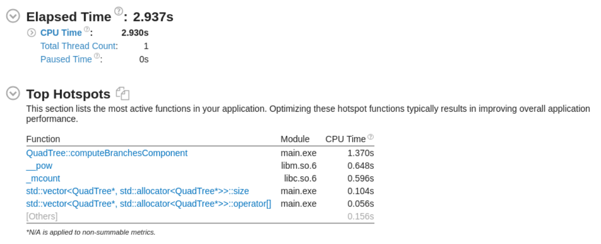
The brute force method has the benefits of being fairly easy to implement. The positions, velocities and accelerations of each particle are initialized on a specific CUDA kernel which allows the data to directly remain on the GPU memory. Ideally, to allow fast computation, the positions and masses of each particles should remain on the shared memory of each block as they are often accessed. However, for both GTX 1070 and Tesla K40 GPUs, the maximum amount of shared memory per block is 49KB. These three arrays were initialized in double precision for precise computation, which maximizes the number of particles to $\frac{49000}{8*3} \approx 2042$ particles which is very small. Hence such optimizations with shared memory usage were not viable.

2.3 Sequential barnes-hut approximation

The barnes-hut approximation is more complex to implement than the brute force method as the tree had to be built and ran through for the force computation.
The quadtree consists of multiple nodes. Each node is represented by a single class called *QuadTree* containing different fields such as the center of mass of the underlying particles. Each node also possess an array of pointers to 4 instances of the same class to represent the children of the current node. With such a data structure, it is possible to construct the quadtree by

looping sequentially on all particles, constructing the tree by adding the particles one by one. Such a data structure comes very handy for a sequential code but prevent the parallelization of the tree building. Indeed, as particles are added one by one and the tree is constructed at the same time, it could induce racing conditions in specific situations, for example when two particles should be added on the same external node at the same time.

Using the `vtune` profiler, the observation that a critical amount of time was spent in a specific line of the method to compute the force component was made. Indeed, the `pow()` function was used which significantly slowed down the computation. Thanks to this observation and a slight change of the *QuadTree* class structure, a non-negligible speedup was introduced as illustrated on Figure 1.



Elapsed Time: 2.937s

CPU Time:	**2.930s**
Total Thread Count:	1
Paused Time:	0s

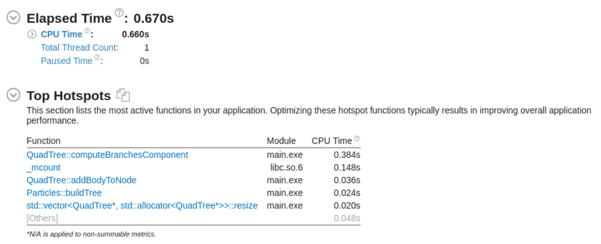
Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	Module	CPU Time
QuadTree::computeBranchesComponent	main.exe	1.370s
__pow	libm.so.6	0.648s
_mcount	libc.so.6	0.596s
std::vector<QuadTree*, std::allocator<QuadTree*>>::size	main.exe	0.104s
std::vector<QuadTree*, std::allocator<QuadTree*>>::operator[]	main.exe	0.056s
[Others]		0.156s

**N/A is applied to non-summable metrics.*

(a) With the `pow()` function.



Elapsed Time: 0.670s

CPU Time:	**0.660s**
Total Thread Count:	1
Paused Time:	0s

Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	Module	CPU Time
QuadTree::computeBranchesComponent	main.exe	0.384s
_mcount	libc.so.6	0.148s
QuadTree::addBodyToNode	main.exe	0.036s
Particles::buildTree	main.exe	0.024s
std::vector<QuadTree*, std::allocator<QuadTree*>>::resize	main.exe	0.020s
[Others]		0.048s

**N/A is applied to non-summable metrics.*

(b) Without the `pow()` function.

Figure 1: Illustration of the speedup gain thanks to the vtune profiler. A speedup of $\frac{2.937}{0.670} \approx 4$ was obtained for a problem size of 4096 particles.

2.4 Barnes-Hut with CUDA

Parallelizing the barnes-hut method was complex as it required a total change of the data structure compared to the sequential code. In general, pointer chasing operations are not optimal on CUDA kernels. It is known that GPUs perform best with array-like data structures. Hence, particular efforts were performed to enable such data-structure for the tree building and force computation.

The goal was to "flatten" the tree to be able to store it in an array for optimal performance on the GPU. The new data structure is an array of a new class called *Node*, but this time the *Node* class does not contain any more pointers to other *Node*. Indeed, the tree structure was implemented using offsets to identify the children of a node in the array. An example of the transformation to the new data structure is illustrated on Figure 2 and 3.

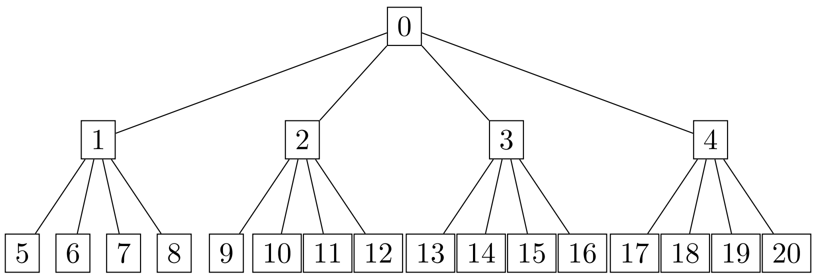


Figure 2: Sequential data structure for a tree of depth of 3

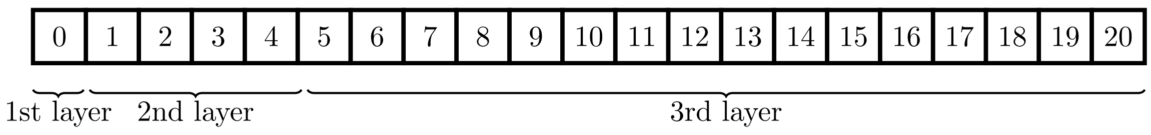


Figure 3: Parallel data structure for a tree of depth of 3

Each deeper layer becomes 4 times bigger than the previous layer (for a *quadtree*) and the children are accessed with offsets.
This data structure enables the possibility of using it on the GPU but it does not solve the problems of racing conditions if one would like to parallelize the filling of this structure. Hence, the tree was built on the host and then sent to the GPU for force computation. Moreover, the size of the array was fixed to embed a maximum depth of the tree.

Finally, the force computation did not embed any particular optimizations. Indeed, a descent along the tree is difficult to vectorise. Moreover, the encountered problem when implementing the brute force method of the shared memory being too small to store the particles information was still present. Thus, the optimization was mostly focused on parameters tuning as will be explained in next sections.

The code for all 3 implementations can be found on c4science at the address:
`https://c4science.ch/source/project_barnes_cuda.git`
`ssh://git@c4science.ch/source/project_barnes_cuda.git`.

3 Prior results

The different codes were run on the Deneb cluster of EPFL, using the GPU nodes and were also run on my personal computer. These two machines present the following characteristics:

Deneb Cluster

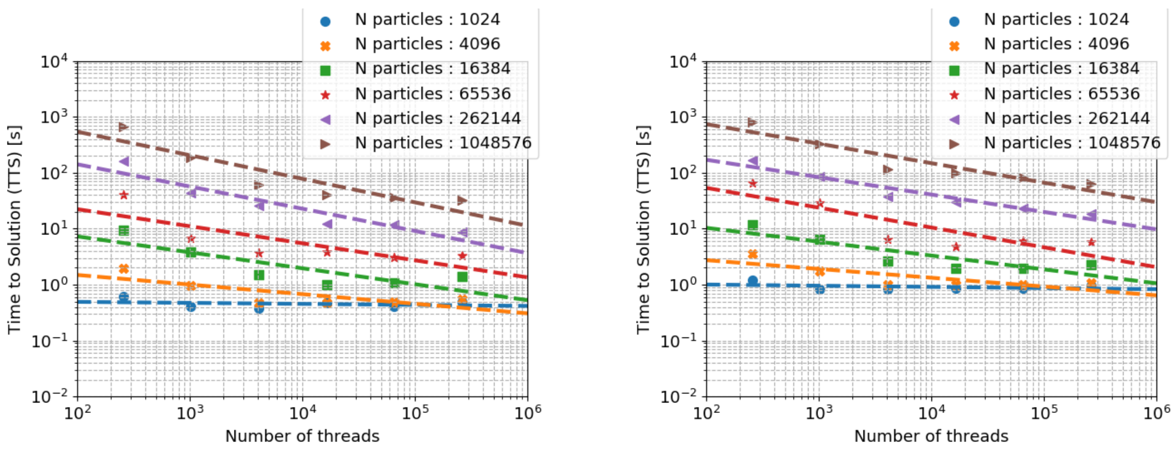
- 376 compute nodes each with 2 Intel 8-cores Ivy-Bridge CPU running at 2.6 GHz, with 64 GB of DDR3 memory
- 16 GPU accelerated nodes, each with 4 K40 NVIDIA cards
- Total peak performance: 293 Tflops/s

Personal Computer

- One Intel Core i7-8700k 6-cores Coffee-Lake CPU running at 3.7GHz stock, OC at 4.8GHz
- One Nvidia GTX 1070
- Total theoretical peak performance: 7.42 Tflops/s

The following analysis is focused on the parallel implementation of the barnes-hut approximation.

First, an investigation on the potential of the developed implementation was tested to verify if it could scale. Indeed, the goal was to deduce the optimal grid and block size for a given number of particles. Results are illustrated on Figure 4. A first observation can be made. The first step to get maximum performance was to spawn as much threads as there was particles. Indeed, if the latter requirement was not fulfilled, different threads had to compute the force component for multiple particles, inducing a longer time to solution. Moreover, if more threads were spawned compared to the number of particles, the spare threads were just idling. As they were still spawned and hence allocated resources, it even sometimes induced a drop of performance.



(a) GTX 1070 (b) Tesla K40

Figure 4: Time to solution (TTS) of the barnes-hut approximation for different problem size in function of the number of spawned CUDA threads, with their linear regression curve. The number of iterations was fixed to 100, the maximum depth of the tree to 10 and the θ parameter to 0.5.

From this point, the following development was implemented on the DENEB cluster only to represent the potential of this application at production scale. Indeed, the Tesla GPUs were built for computation while the GTX GPUs were built for gaming.
GPUs are rarely shared between different jobs. In the case of this project, the main goal was to maximize the performance of the developed application for a given problem size. Hence, the following sections are devoted to the optimization of solving the NBody problem using the barnes-hut approximation on a single node.

3.1 Roofline Analysis

Using different properties of the `cudaGetDeviceProperties()` function, the Memory Bus Width and the Memory Clock Rate could be extracted and multiplied together to get the theoretical peak memory bandwidth (with a factor 2 due to DDR memory, without ECC). For the peak performance in double precision, different values considering the hardware should be known. Indeed, the theoretical peak performance is known to be (for double precision) [Frequency]*[# Double Precision cores]*[FMA]. The GPU clock rate could also be extracted using `cudaGetDeviceProperties()`. For the number of double precision cores, the value was found online [1]. For the Tesla K40, the resulting values are illustrated in Table 1. These values define

[1]Number of double precision cores was found here `https://videocardz.com/69378/nvidia-announces-tesla-v100-with-5120-cuda-cores`

the theoretical roofline curve illustrated in Figure 5.

Memory Bus Width	Memory Clock	GPU Clock	# DP Cores	Peak Performance	Memory Bandwidth
384 Bits	3.004 GHz	745 MHz	960	1.43 Tflops/s	288.384 GB/s

Table 1: Specifications for the Tesla K40 GPU card

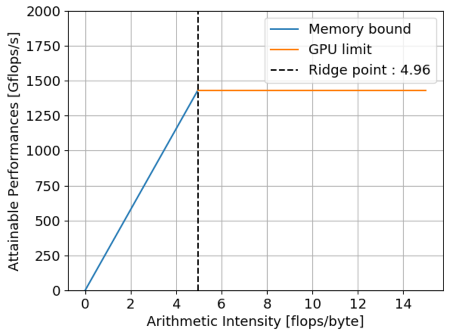
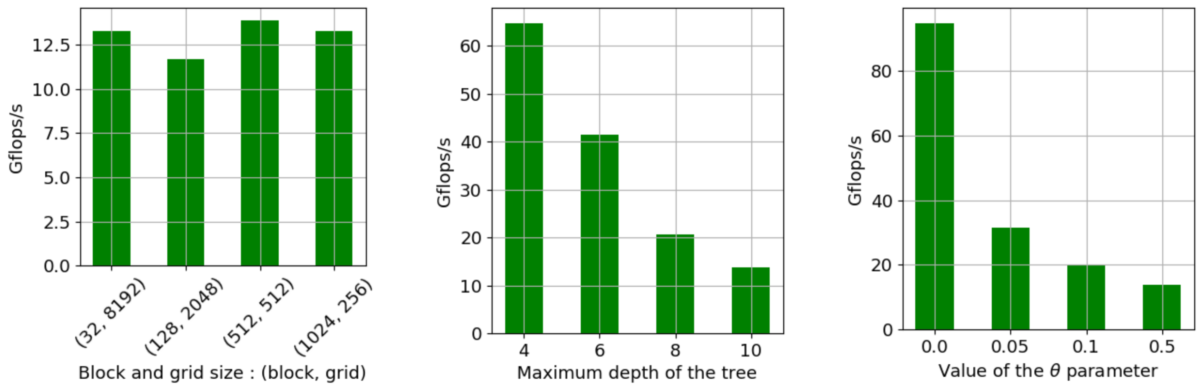


Figure 5: Theoretical roofline curve for the Tesla K40

The critical kernel for this application was the kernel computing the force components and updating the pose of the particles. To measure the Gflops/s of this kernel, the `nvprof` profiler was used. Indeed, this profiler enables the measure of the average number of floating point operations (single and/or double precision) per iteration on the kernel. It also enables the measure of the average time spent per iteration on the kernel. Knowing these two values, the obtained Gflops/s were computed as $Gflops/s = \frac{flops}{t}\frac{1}{10^9}$. The effect of different parameters on the performance in terms of Gflops/s are represented in Figure 6.



(a) Influence of the grid and block size. Mayimum depth of the tree was fixed to 10 and θ to 0.5.

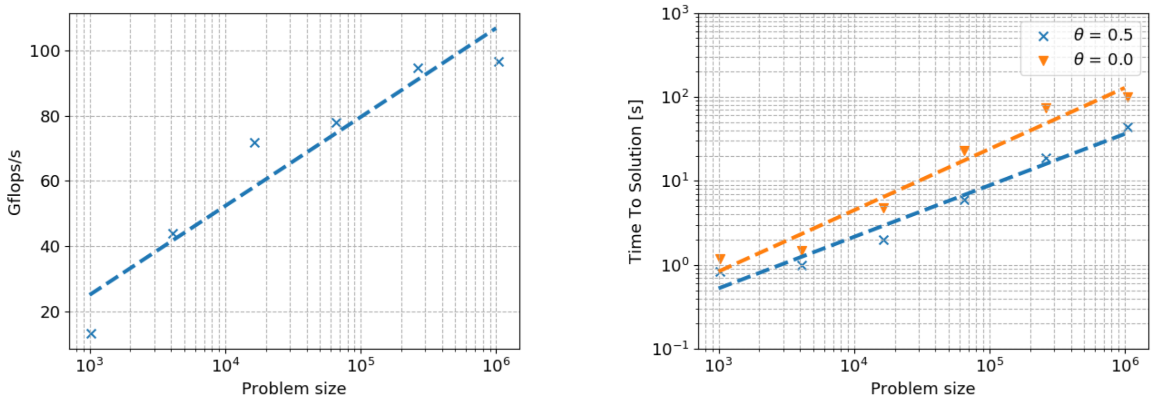
(b) Influence of the maximum depth of the tree. Block and grid size were fixed to 512 and θ to 0.5.

(c) Influence of the θ parameter having fixed the grid and block size to 512 and the depth to 10.

Figure 6: Influence of different parameters to the performed Gflops/s by the parallel implementation of the barnes-hut approximation. These results corresponds to a problem size of 262144 particles.

Observing Figure 6 (a), one can state that the repartition of the grid and block size do not affect drastically the performance. However, the higher Gflops/s were observed having the same size of grid and block (512 and 512). About Figure 6 (b), it can be observed that the smaller the maximum depth of the tree, the higher the Gflops/s which can be explained by the following: looking at the different Gflops/s values and matching them on Figure 5, one can see that the program is strongly limited by the memory bandwidth. By reducing the size of the tree, the number of memory accesses are diminished which directly increases performance. Finally, Figure 6 (c) illustrates that the smaller the θ parameter, the higher the Gflops/s probably due to the increase number of calculations. Indeed, the smaller θ, the deeper the algorithm has to go in the tree hence inducing more computations.

These graphs represent the influences on performance of these parameters one at a time considering all the others fixed. When optimising different parameters at the same time, the resulting influence on the performance may not be only positive. In fact, quite the contrary was observed as the best performance were obtained with maximizing the depth of the tree and minimizing the θ parameter. If possible (less than 1024^2 particles), the grid and block size should be set to the same value. Results for different problem size with this configuration are illustrated in Figure 7 (a). One can state that the bigger the problem size, the higher the Gflops/s.



(a) Maximum obtained Gflops/s with respect to the problem size.

(b) Time to solution with respect to the problem size and the θ parameter.

Figure 7: Maximum obtained Gflops/s by using the optimal parameters developed in Section 3 with respect to the problem size. The (b) figure illustrates the TTS in function of θ. For both graph, the maximum depth of the tree was fixed to 10. These results were measured on the Tesla K40.

This configuration of parameters may have induced the higher Gflops/s, but the time to solution also increased due to the minimization of the θ parameter as illustrated on Figure 7 (b). One should hence tune the parameters differently when performance or speed is needed. For fast computation, the value of θ should be in the range $[0.5, 1]$.
Observing Figure 8 which represents the same plot as Figure 7 (b) but with the two other implementation also illustrated for comparison, one can state that, for very small problem size, the sequential implementation of the NBody solver was the fastest. From ≈ 7000 particles, the

parallelization of the barnes-hut approximation becomes the fastest. The brute force solver is overall slower than the sequential or parallel barnes-hut solvers. The brute force results were fitted using a polynomial of degree 2 to better represent the data and the $\mathcal{O}(n^2)$ complexity. Finally, as a parenthesis, Figure 9 illustrates the power of latency in HPC.

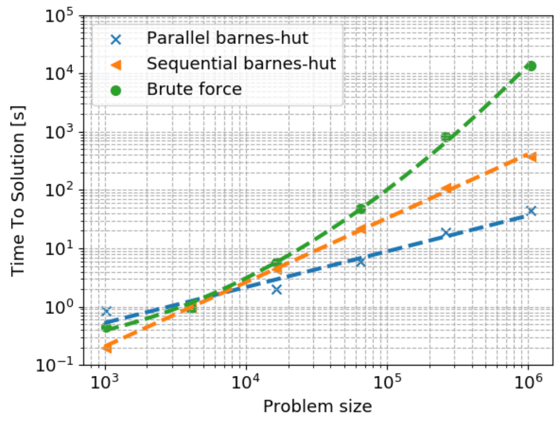


Figure 8: Comparison of the TTS for the three different implementations. For the barnes-hut curves, the depth of the tree was fixed to 10 and the θ parameter to 0.5.

4 Resources budget

In order to fulfil the requirements of our project, we present hereafter the resource budget.

4.1 Computing Power

Using the Figure 7 and targeting a problem size of $2^{24} \approx 17M$ particles, one can extrapolate that, when using the application for performance, approximately 140.23 Gflops/s would be obtained. For fast computation, this problem should be solved in approximately 200 seconds when fixing the θ parameter to 0.5. These results are considering a maximum depth of the tree of 10 and the computation of 100 iterations.

In terms of memory, there are several variables that have to be stored per particle describing its motion (acceleration, velocity, position and mass). For 2^{24} particles in 2D, this amount is approximately 939MB. Such a number is not a problem for modern GPUs including the Tesla K40.

4.2 Raw storage

The application source codes and executable weight approximately 741KB. When the debug mode is not activated, the code does not output anything hence no more space is needed. In debug mode, the positions of each particle are stored in a text file after each iteration. For big problem size, this file can weight multiple 100s of MB.

4.3 Grand Total

Total number of requested cores	2^{24}
Minimum total memory	1GB
Maximum total memory	10GB
Temporary disk space for a single run	741KB (without save)
Permanent disk space for the entire project	10GB
Communications	CUDA
License	own code (BSD)
Code publicly available ?	yes
Library requirements	None
Architectures where code ran	Tesla K40

5 Scientific outcome

Computing the solution to the NBody problem can have multiple benefits in different fields. As stated in [1], solving this problem in an efficient way could improve the performance of computer graphics. As stated in the abstract, this problem suits particularly well the movements of celestial bodies. Solving this problem efficiently could allow to solve bigger problems than what we are capable now in a reasonable amount of time.

References

[1] Jaswinder Pal Singh, Chris Holt, Takashi Totsuka,Anoop Gupta and John L. Hennessy*Load Balancing and Data Locality in Adaptive HierarchicalN-body Methods: barnes-hut, Fast Multipole, and Radiosity*, Computer Systems Laboratory, Stanford University

[2] Scientific IT and Application Support, `http://scitas.epfl.ch`, 2015

A Additional Figures

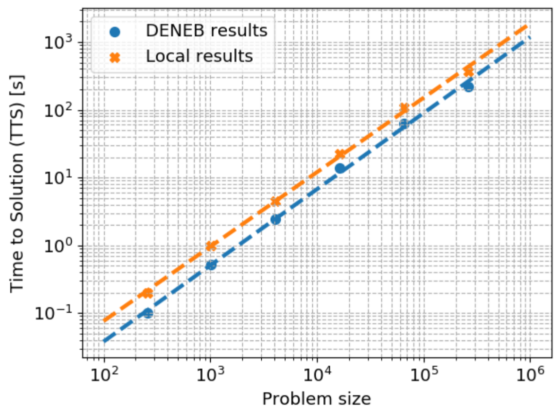


Figure 9: Illustration of the power of latency for the sequential barnes-hut program. The Ivy-Bridge CPU of the cluster runs at 2.6GHz while the local i7-8700k runs at 4.8GHZ, giving a difference of frequency of roughly 2. This directly results as a difference very close to 2 in terms of TTS.