Parallel framework for Smooth Particle Hydrodynamics

B.Tech Project
Stage I

by

Pushkar Godbole

09D01005

under guidance of

Prof. Prabhu Ramachandran

Department of Aerospace Engineering
Indian Institute of Technology - Bombay
Mumbai

December 2012

Certificate

This is to certify that this B.Tech Project - Stage I report titled **Parallel framework for Smooth Particle Hydrodynamics** by Pushkar Godbole (Roll no. 09D01005) has been examined and approved by me for submission.

Date : Name and signature of guide

Acknowledgements

I would like to thank my guide Prof. Prabhu Ramachandran for taking time out of his busy schedule to keep a check on my work and for providing invaluable guidance and support. I would also like to thank Kunal Puri for being an implicit co-guide, for all his assistance and patience throughout the semester. The regular meetings and discussions with Prof. Prabhu and Kunal have been really gratifying. Working on the project has been fun and a great learning experience more so. Looking forward to the Stage II of my B.Tech Project.

Pushkar J. Godbole

Abstract

Smooth Particle Hydrodynamics (SPH) is a Lagrangian, mesh-free method for the numerical solution of partial differential equations. Evolution of the particle properties in accordance with the governing differential equations describe the flow. An accurate description of the flow requires a large number of particles (a typical 3D dam break problem can have half a million particles). Hence parallel paradigms need to be explored for any realistic implementation. With the the basic structure to solve SPH problems already in place, in form of PySPH (Python framework for Smooth Particle Hydrodynamics), an efficient parallel module to augment and accelerate the solver is needed. This report will detail and evaluate the concepts and implementation methodologies for such a parallel module.

Keywords : Python, PySPH, Parallel computing, mpi4py

Contents

List of Figures

1 Introduction

Smoothed Particle Hydrodynamics (SPH) is a particle based computational simulation technique. It is a mesh-free Lagrangian method (where the coordinates move with the fluid), and the resolution of the method can easily be adjusted with respect to **particle properties** such as the density. The variation of these particle properties based on the underlying differential equations leads to evolution of the flow in time.

1.1 SPH

The method works by dividing the fluid into a set of discrete elements, referred to as particles. These particles have a spatial distance known as the **"smoothing length"**, over which their properties are "smoothed" by a kernel function. This means that the physical quantity of any particle can be obtained by summing the relevant properties of all the particles which lie within the range of the kernel.
A **kernel function** is essentially a function which defines the variation of the influence of a particle on its neighbours with variation in the distance. Kernel functions commonly used include the **Gaussian function** and the **cubic spline**. The latter function is exactly zero for particles further away than two smoothing lengths unlike the Gaussian function, in which the influence follows the Gaussian curve and every particle maintains a diminishing but finite impact with the distance.
PySPH is an open source framework for Smoothed Particle Hydrodynamics (SPH) simulations. It has been developed by Prof. Prabhu Ramachandran, Kunal Puri, Pankaj Pandey and Chandrashekhar Kaushik of Aerospace Engineering Department, IIT-Bombay. The current working version of PySPH is capable of carrying out simulations in serial and parallel. But the present implementation of the parallel module is not sufficiently efficient and hence ends up being slow (slower than the serial implementation). To solve large and realistic (million particle) problems, PySPH needs an augment of an effective parallel module. Further investigation and possible re-implementation of this parallel module is thus necessary. The report further will detail this analysis and propose an outline of the line of action for the implementation.

1.2 Parallel SPH

The two primary tasks in SPH, neighbour-particle location and force computation both require an order n^2 time at best. Therefore with increase in the number of particles, the run time of a serial implementation is bound to grow at least quadratically. It is thus evident that a parallel implementation is inevitable. A parallel implementation of the SPH solver would essentially run simultaneously on multiple processors, while each processor solves one portion of the entire problem.
The primary idea can be briefly explained as follows :
The particles are equally distributed amongst processors. Each processor is assigned a set of particles to evolve (called local particles). But only the information about the local particles would not suffice. Because at the problem-defined geometric boundaries of every processor, the particles need to interact with the particles in the neighbouring processors as well. Thus an extra halo region comprising of particles near the boundaries (called ghost particles), needs to be exchanged between the neighbouring processors.

What sets the SPH method apart from fixed grid Eulerian approaches is that the particles are free to move all over the computational domain which is often free space. It must therefore be noted that as the problem evolves, the particle-properties and particles themselves in the halo region are bound to change. Hence at every time step, the halo region in each processor too needs to be updated.
As the prime aim of SPH is solving the problem, the parallelizer should take trivial amount of time compared to the actual solver. Only then can the parallel implementation be said to work effective.

2 PySPH Overview

The parallel module is to be built upon the PySPH framework. Thus the module needs to be designed such that it would dock with minimal changes made to the current PySPH structure. It must be as de-linked from the actual PySPH solver as possible. Therefore, before moving on to the parallel implementation, consideration has to be given to understanding the structure of the PySPH framework. This chapter will detail the anatomy of PySPH in brief.
Major portion of this section has been referenced from the official PySPH documentation website. The following flow chart explains the basic process of problem evolution in PySPH.

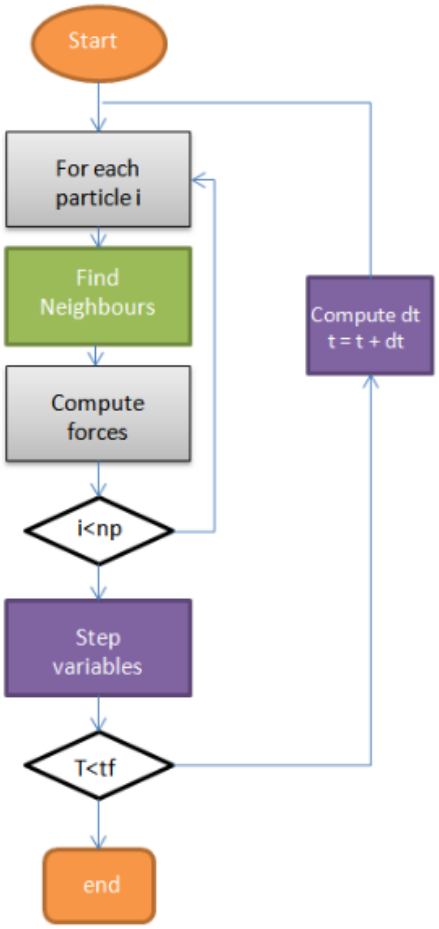

Figure 1: PySPH flow-chart

Here :
i : particle index
np : number of particles
t : current simulation time
tf : final time (total simulation time)
dt : time step

The colours are keyed by the following scheme.

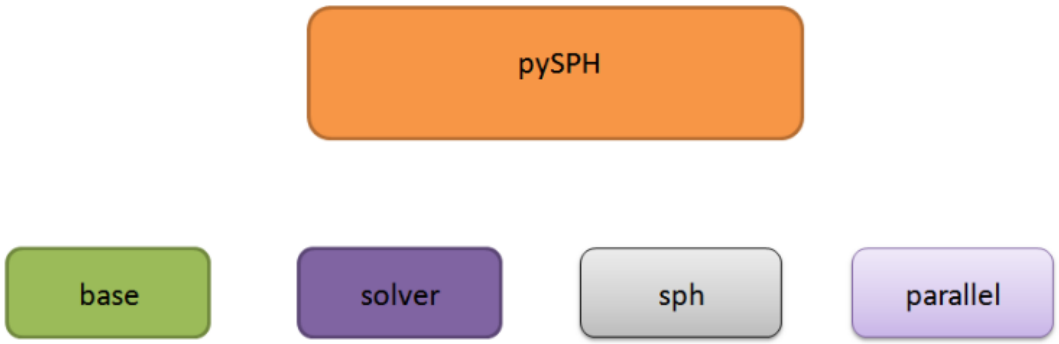

Figure 2: PySPH modules

PySPH attempts to abstract out the operations represented in the flowchart. To do this, PySPH is divided into four modules.

2.1 base

The **base** module defines the data structures to hold particle information and index the particle positions for fast neighbor queries and as such, is the building block for the particle framework that is PySPH.
As seen in the flowchart (PySPH flow-chart), the find neighbors process is within the inner loop, iterating over each particle. Thus the base module is also responsible for locating the neighbours of every particle. This module can be thought of as the base over which all of the other functionality of PySPH is built.

2.2 sph

The sph module is where all the SPH internal functions are defined.

2.3 solver

The solver module is used to drive the simulation via the Solver object and also the important function of integration (represented as step variables in the flowchart: PySPH flow-chart). Other functions like computing the new time step and saving output (not shown in the flowchart) are also under the ambit of the solver module.

With certain modifications and simplifications being made in the current PySPH version, the above structure is subject to change to some extent soon.

2.4 parallel

It may be noted from the PySPH modules block diagram that the parallel module is shown to be independent of the base, sph and solver. The parallel module will essentially sit on top of other modules and provide subproblems to the solvers across all processors thus de-linking itself from the actual SPH simulation. The parallel module will handle the responsibility of efficiently allotting particles and exchanging the updated particle information amongst processors promptly. Thus the solver need not “see” the parallel module and vise-versa.

3 Parallelizer

The parallelizer has to be de-linked from SPH simulation, so that the application scripts (written by end-user) are independent of the type of run (serial/parallel). Then the end-user would only need to provide the problem description and choose the type of run without worrying about the internal implementation. The parallelizer therefore has three primary tasks to accomplish efficiently :

- Achieve and maintain a balanced workload (in terms of local particles) across processors
- Minimize the the halo region (number of ghost particles) that needs to be exchanged between neighbouring processors
- Achieve prompt exchange of ghost particles at every time step

The first task will ensure that all the work is fairly distributed across processors thus maintaining an effective utilization of resources. Since the ghost particles need to be exchanged at every time-step, minimizing the data transfer required amongst neighbouring processors is very crucial. Finally, with the ghost particles determined, these particles need to be exchanged speedily. Hence an adept protocol needs to be implemented to achieve the data transfer. The time consumption of these parallelization tasks needs to be trivial compared to SPH solver for a meaningful parallel implementation of PySPH.
It must be noted that, the ghost particles from the neighbouring processors will affect the local particles but will not be affected by the local particles. Their properties will be updated only during the next iteration when new halo region data is received from the neighbouring processors.

3.1 Load-balancing and Halo-minimization

The parallelizer needs to accomplish a dual task of equally distributing the particles while maintaining a minimal halo region between processors. Randomly distributing the particles would lead to steep rise in the size of the halo region thus raising the data transfer requirement. The most intuitive and effective solution to this problem is equally distributing the particles region-wise according to the problem geometry. This will achieve fair distribution of workload and minimize the data transfer required between processors. The equivalent distribution of workload is called **Load-balancing**.
The size of the halo region depends upon three factors :

- **Thickness of the halo region required**
  The thickness of the halo region is determined by the interaction radius. The thickness of the halo region needs to be twice the interaction radius.
- **Connectedness of the region assigned to a processor**
  With thickness of the halo region fixed by the interaction radius, the volume of the halo region would be minimized by maximizing its connectedness. If a particle totally offset from its major region is allotted to a processor, it would unnecessarily require all particles in the halo of this particle to be transmitted to this processor as ghost particles. Thus, the regions allotted to all processors must preferably be connected and closed.
- **Surface area of the region**
  Now with the thickness known and connectedness maintained, the halo region would

be minimized by minimizing the surface area of the geometric region allotted to the processor. Surface area is a direct determinant of the common region that needs to be shared between processors. Hence the particle allotment will have to be done to maintain optimum connectedness and surface area.

This three-step approach to minimize the data transfer required has been termed as **Halo-minimization**.

But as the problem evolves, due to the Lagrangian mesh-less free-flow property of the SPH method, the particles are bound to travel all through the problem space and end up completely distorting the initially allotted regions of the problem due to "mixing". This will again lead to steep rise in the size of halo region across all processors. The problem can be clearly understood from the following dam-break example. It can be seen that the initial intelligent allotment of particles to the processors has completely been obfuscated by the problem evolution.

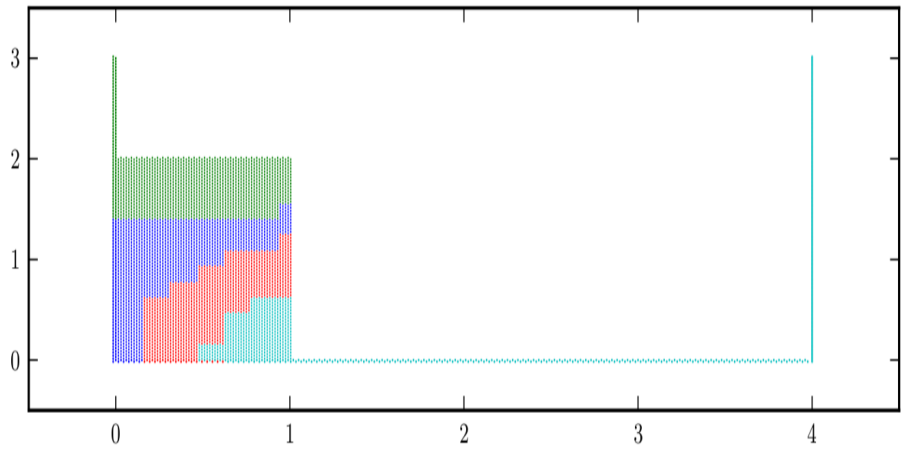

Figure 3: Dam break : Initial partitioning

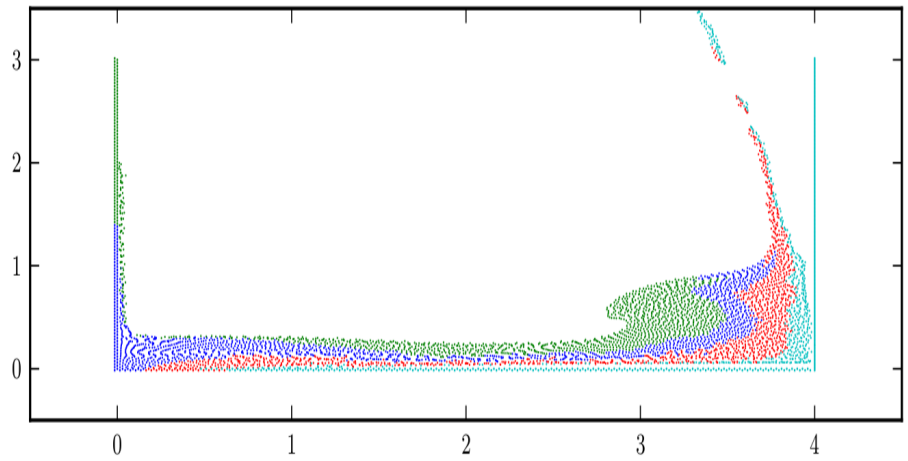

Figure 4: Dam break : Obfuscated partitioning due to problem evolution

Evidently a static particle allotment to processors at the beginning of the run will only harm the the simulation. A dynamic load-balancing is therefore necessary at intermediate intervals of the simulation.

3.2 Load-balancing and Halo-minimization : Implementation

Load-balancing and Halo-minimization are so interrelated that they need to be dealt with as a single problem. Graph partitioning techniques are often used for partitioning mesh based schemes as they fulfill the requirements. These algorithms re-interpret the mesh data structure as a graph with the elements defining the nodes and the element connectivity as the edges between the nodes. The algorithm then proceeds to partition the graph by performing cuts along the edges. The output is a set of well-partitioned closed sub-graphs with approximately equal number of nodes.
While this works in principle, the re-interpreted graph can have too many edges, leading to very expensive running times for the algorithm. An alternative is to use the particle indexing structure as an implicit mesh where the cells define the nodes and the cell neighbours (27 in 3D and 9 in 2D) define the node connectivity. The graph is now invariant to the number of neighbours for a given particle (resolution in SPH). Nodes can be weighted based on the number of particles in the corresponding cell to achieve a good load balancing across processors. Below is a comparison between the two techniques in terms of **time required for partitioning** and **quality of partitioning**.

3.2.1 Node partitioning vs Cell partitioning

Various open-source implementations of graph-partitioning algorithms exist. Metis is one such package. Additionally it also has a Python wrapper by the name PyMetis. The main advantage of PyMetis is that it is extremely light-weight and trivial to implement. PyMetis has been used to partition the following problem space and evaluate the results. n particles were distributed randomly in a 10x10 square. PyMetis was used to partition the problem space into 10 chunks.
Two aforementioned methods were compared :

Node partitioning :
Feed the particle distribution directly to PyMetis. PyMetis reinterprets the distribution as a graph and splits the graph into 10 chunks. The output is, 10 independent sets of well-distributed particles.

Cell partitioning :
Bin the particles into implicit cells, based on their geometric position in the problem space. Feed the implicit cells containing particles to PyMetis along with cell weights (number of particles in each cell). PyMetis reinterprets the cell distribution as a weighted graph and splits the graph into 10 chunks. The output is, 10 sets of cell groups with approximately equal number of particles in each group.

3.2.1.1 Partitioning time :

The Node partitioning technique is bound to take more time compared to the Cell partitioning technique owing to more nodes and effectively more cuts needed for partitioning. The following plot compares the times taken by the two methods w.r.t. number of particles. The Cell partitioning has been evaluated for varying cell sizes. The cell sizes in the plots are measured in terms of the Interaction Radius (IR).

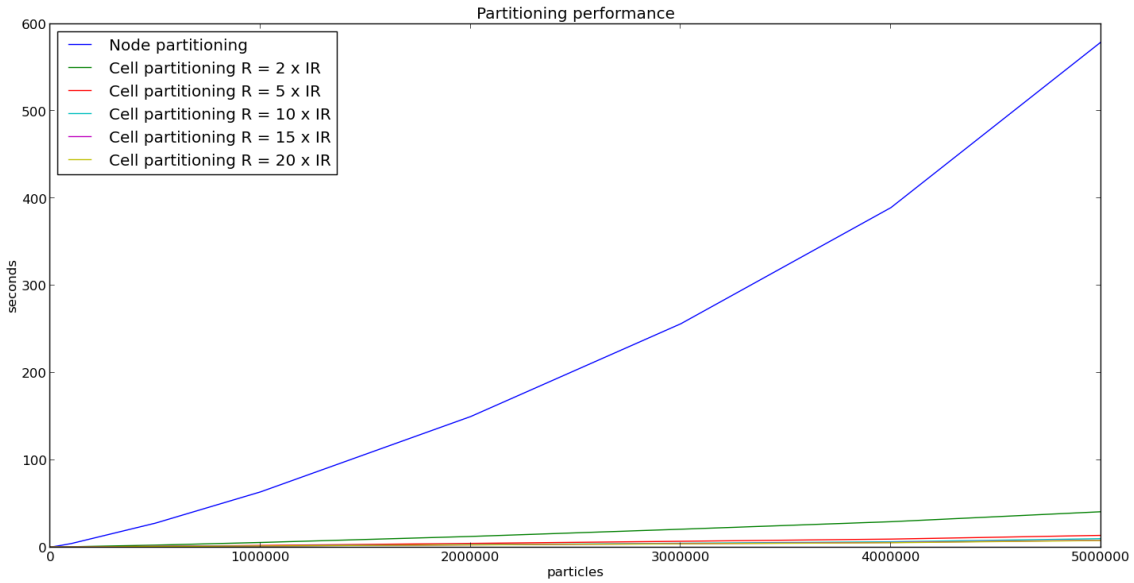

Figure 5: Partitioning : Time performance

It can be seen that Node-partitioning is taking extremely long compared to Cell-partitioning. Even compared to the Cell-partitioning results with cell size twice of IR, Node-partitioning takes almost 15 times longer. Also the growth has more of a quadratic trend than linear as the number of particles increases. Node-partitioning is therefore rendered a completely infeasible option to implement graph partitioning.
The following graph more clearly depicts the performance of Cell-partitioning at various cell-radii. Evidently, as the cell size increases, the partitioning takes lesser time to execute.

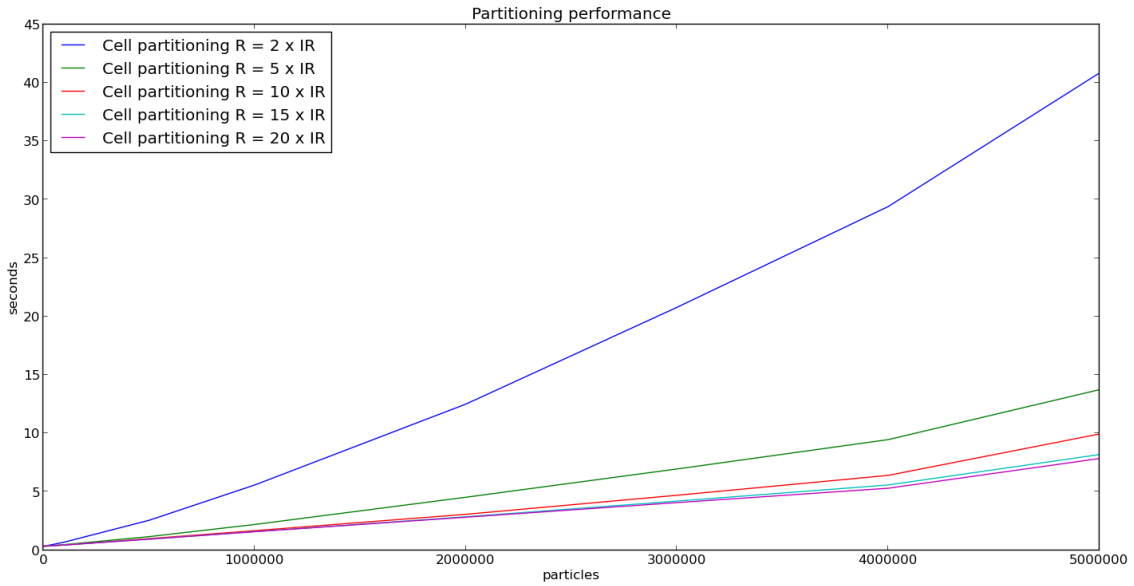

Figure 6: Cell partitioning : Time performance

It must be noted that although the time-performance of Cell-partitioning improves with increase in the cell size, the added advantage of increasing the cell size diminishes further up. Also care needs to be taken to strike a right balance between time-performance and

quality-performance of the partitioning. The next subsection on partitioning quality will clarify this better.

3.2.1.2 Partitioning quality :

Although the Cell partitioning takes lesser time than Node partitioning, the quality of particle distribution across all chunks needs to be judged. Larger Cell size will lead to coarser particle distribution. An optimal balance needs to be reached between partitioning time and partitioning quality.
An ideal particle distribution will have equal number of particles in every chunk. The deviation from ideal distribution can be quantified in terms of the deviation from this mean *(Total no. of particles/Number of chunks)*. Standard deviation of the particle distribution across all chunks is plotted in the graph below for various cell-sizes in terms of interaction radius.

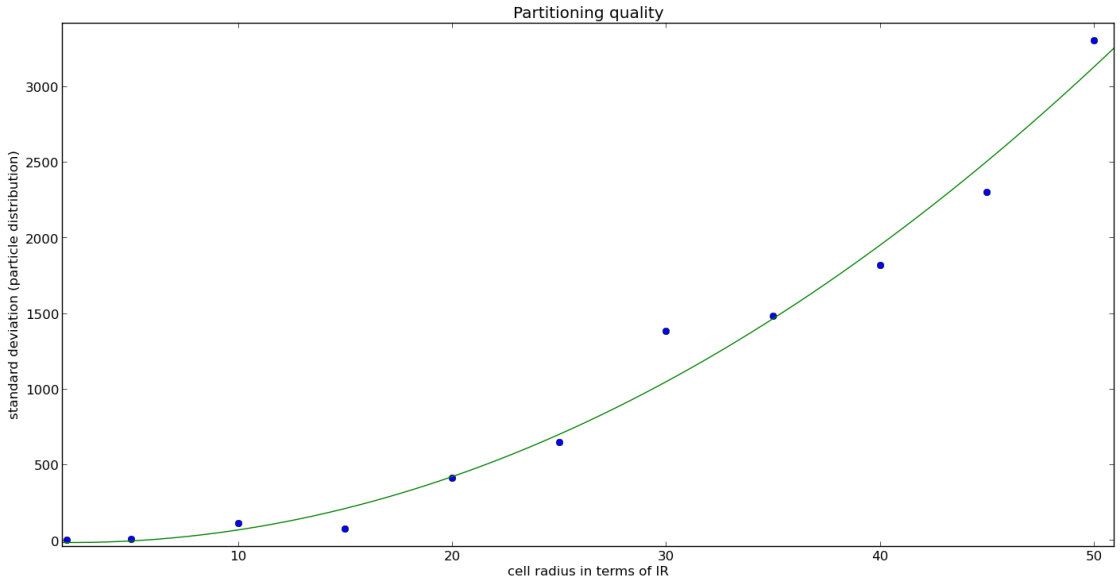

Figure 7: Cell partitioning : Quality performance

It can be seen that the standard deviation increases quadratically with cell size. At cell radius of $50 \times IR$ the standard deviation has crossed 3000. This means that some chunk has 3000 particles more/less than the mean and hence is handling an unbalanced load.
Prima-facie, cell-radius of $15 \times IR$ appears to be a good choice for the cell size both in terms of partitioning time and partitioning quality. Although no definite claims are being made at this stage.

3.2.2 Zoltan

Like PyMetis, **Zoltan** is another toolkit that employs various parallel combinatorial algorithms for unstructured and/or adaptive computations. Besides graph-partitioning Zoltan facilitates use of other techniques like Recursive Coordinate Bisection (RCB), Space-Filling Curve (SFC) and Refinement Tree Based Partitioning techniques. These algorithms require as input, coordinates of objects to be partitioned. For a particle method like SPH, the particle positions serve as the most natural input. Although simple in their approach, the relative performance of geometric and graph based partitioning algorithms needs to be

evaluated before drawing any conclusion. Besides Zoltan also provides support for other popular graph partitioners like Metis, ParMETIS and Scotch.
The performance is assessed in terms of the total time spent during the simulation which includes overhead in determining the halo region for a given partition. The results are readily applicable to SPH and more generally, to any Lagrangian scheme where the particles/cells are free to move in computational domain. Zoltan is implemented in C++. A python wrapper for the same is being developed by Kunal Puri. Since Zoltan specifically claims particle methods to be one of its applications, we propose to further explore and positively implement it in the parallel module.

3.3 Data transfer

With Load-balancing done and ghost particles determined, the next task is to actually transfer the particle data across processors. This has to be done speedily so as to not hinder the actual simulation. Data transfer must not become a bottle-neck for the simulation to proceed. Hence an efficient method to transfer the data has to be implemented.
Open MPI being the most widely used cross-platform Message Passing Interface, is the most appropriate choice for the task. Additionally, Open MPI also has a Python wrapper by the name **mpi4py** which further eases the task. This section will mainly evaluate various data transfer methods provided by mpi4py w.r.t. our needs. Finally it will conclude with a justified choice of a method.

3.3.1 Open MPI/mpi4py Overview

As mentioned above, MPI is a parallel Message Passing Interface. A typical parallel implementation using open MPI initiates multiple processes. These processes run independently and communicate with each other throught MPI.

MPI has various methods to effectuate the messaging. These can be broadly classified into two categories :

Point-to-point communication :
This is the simplest message passing technique and involves one sender and one receiver. The sending process knows the receiving process and visa-versa. The receiving process waits until it receives the data from the sending process. A unique passage is used by the two processes in point-to-point communication. This type of communication is mainly useful when specific processes need data from other specific processes and there is no particular root process which sits on top of and handles all other processes.

Collective communication :
This technique is slightly complex and involves multiple processes at a time. It is used when the root process has to receive data from all other children processes or visa-versa. Thus it is a one-to-many and many-to-one communication. The technique is specifically helpful when the root does not change frequently in a run and co-ordinates the entire run with all other children processes.

With the message passing method chosen, MPI facilitates usage of two protocols to transfer the data. An appropriate protocol must be chosen according to need. The two protocols are briefly described below :

Pickling :
This is a very simple and trivial-to-implement protocol. This protocol basically takes the data to be sent and encodes it using Python's **Pickling** module. The pickle is then sent across the processors. The receiving process again un-pickles the data at the end of the transfer. The pickling protocol accepts all standard python data-structures (variables, lists, dictionaries, tuples). Although simple, Pickling has a disadvantage. The pickling and un-pickling consume excess time over and above the data transfer thus slowing down the entire process. Pickling protocol should therefore be used only when voluminous data is not to be sent and frequency of data transfer is not very high. It is not an efficient means to transfer data.

Buffers :
This protocol is highly efficient but difficult to implement. It does away with the encoding and decoding process and directly sends the raw data across, as buffers. It uses Python's numpy arrays as buffers to accomplish the transfer. The implementation is made complex by the fact that, the receiver needs to know the size of the data it intends to receive from the sender before hand. The receiver is required to define the receive-buffer appropriately before the sender sends the send-buffer. Due to elimination of pickling, the process is highly efficient and recommended when there is frequent and voluminous data transfer involved.

3.3.2 mpi4py Methods

Various requirement specific data transfer methods exist in mpi4py. These methods fall in one of the two categories each mentioned above. This sub-section will briefly describe the relevant methods that have been tested and evaluated for performance.

send & receive :

This is a pickling method, simplest of all to implement but also the slowest. Any basic python data-structure (lists/tuples/dictionaries) can be sent directly using this method. The data to be sent is internally pickled and then sent as one long chain. Once received, it is un-pickled by the receiving processor and converted to original state. This is a point to point communication method where there is a one-to-one link between the two interacting processors.

Send & Receive :

This is a buffer based method where the objects sent can only be numpy arrays (even if the data is a single variable). Also the method requires the array sizes to be exact. i.e. an array expecting to receive data from a processor should be of the exact size as the length of the buffer to be received. Thus a two step process, first exchanging the length of the incoming buffer and then actually receiving the buffer is required. This is also a point to point communication method. The method is significantly (3-4 times) efficient than the aforementioned send & receive method.

Scatterv :

This is also a buffer based method and can only deal in numpy arrays. Unlike the previous method, it is collective communication method. As the name suggests, the function scatters data from the root node to all other nodes in the run. It differs from the Scatter function in that, it can send variable amount of data (this also includes no data) to different processors. (So a processor in the PySPH run can allocate a pre-decided chunk

of its output buffer to a particular processor.) It also requires 2 step communication like the Send & Receive method. This is achieved by using the Broadcast function which is the same as Scatter function except that it sends a single variable to all other processors. Once the data size is known the receive array can be defined and then the data received in the second step.
This method is specifically suitable for codes where the root node remains fixed. But in case of PySPH, every node is a root node and scattering the data simultaneously. The method was found to be robust and perfect until the root node was kept fixed. When implemented for variable root-node case, the run crashed. This crash may be explained as follows :
Consider only 2 processors. When the code starts, both processors start the run. Processor 0 broadcasts its own data-size. After receiving the data-size from Processor 0, Processor 1, broadcasts its own data-size. Then they define their own receive arrays and proceed to scatter their data. But all these processes need not occur simultaneously unless a comm.Barrier() (comm.Barrier() blocks all processes until every other process reaches that point in the run) is placed at every point in the code. The Processor 0 essentially expects Processor 1 to scatter its data only after it has defined its receive array which in turn requires Processor 1 to send its data-size prior to that. The method inherently expects synchronicity of all the processes which is far from reality. Thus a method where every process can run independently of each other is necessary. Gatherv is one such method.

Gatherv :

It is a buffer based method like Scatterv. In this method the root node gathers data from all other processors. Similar to the scatter function, it also requires the gathering process to know the amount of data is to be received from every other process. This is achieved using the Allreduce function which is a synchronous function and reduces the values in a particular variable to a single value subject to some operator.

Ex)
P0 : [3, 1, 0]
P1 : [0, 2, 0]
P2 : [0, 0, 4]
Apply Allreduce with MAX as the operator. Now,
P0 : [3, 2, 4]
P1 : [3, 2, 4]
P2 : [3, 2, 4]

Now all processors have the data-size information to be expected from every other processor. Now they can define their own receive array and gather data from the other processors. The primary difference between this and Scatterv is that every process depends on itself. The data to be gathered from the other processes is already present there. Since the defining of the receive array and gathering of data occur on the same process there is no requirement of synchronicity.

3.3.3 Performance analysis

Data transfer across processors was gauged. The test case used was :
Initiate 4 parallel processes using mpi4py. Each process sends its own data to other 3 processes.
The time-performance of send & receive, Send & Receive and Gatherv was evaluated using this test case.

Test data-transfer with increasing number of particles :
Particles here are represented by single float variables. Performance wise Send & Receive, Scatterv and Gatherv all perform equally well. send & receive is extremely slow as anticipated initially due to pickling.
Graph below shows the comparative performance of the methods :

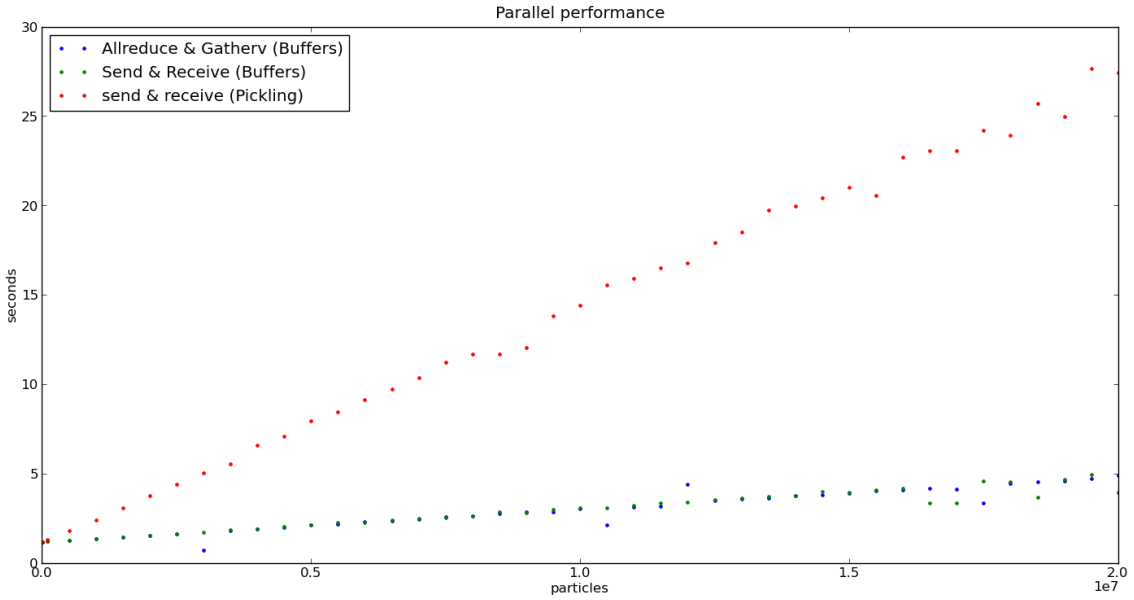

Figure 8: mpi4py : Parallel performance

Test performance of PySPH's __reduce__ method :
All particle properties were transferred during this exchange. PySPH's __reduce__ method sends the pickle of the entire dictionary containing all particle properties. send & receive sends pickles of individual properties while, Send & Receive sends buffers of individual particle properties. PySPH's __reduce__ function turns out to be the slowest.
Following graph depicts the comparison :

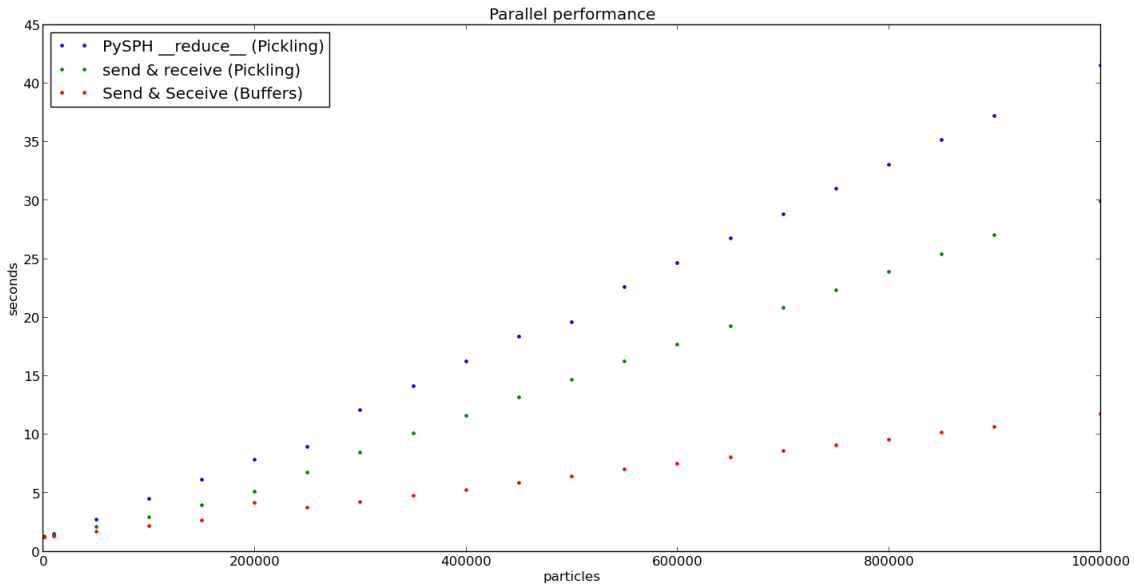

Figure 9: PySPH : Parallel performance

Test performance of above methods for multiple queries :
All particle properties of 5000 particles were transferred using the three methods PySPH's __reduce__, send & receive and Send & Receive. Similar trend was observed in this case :

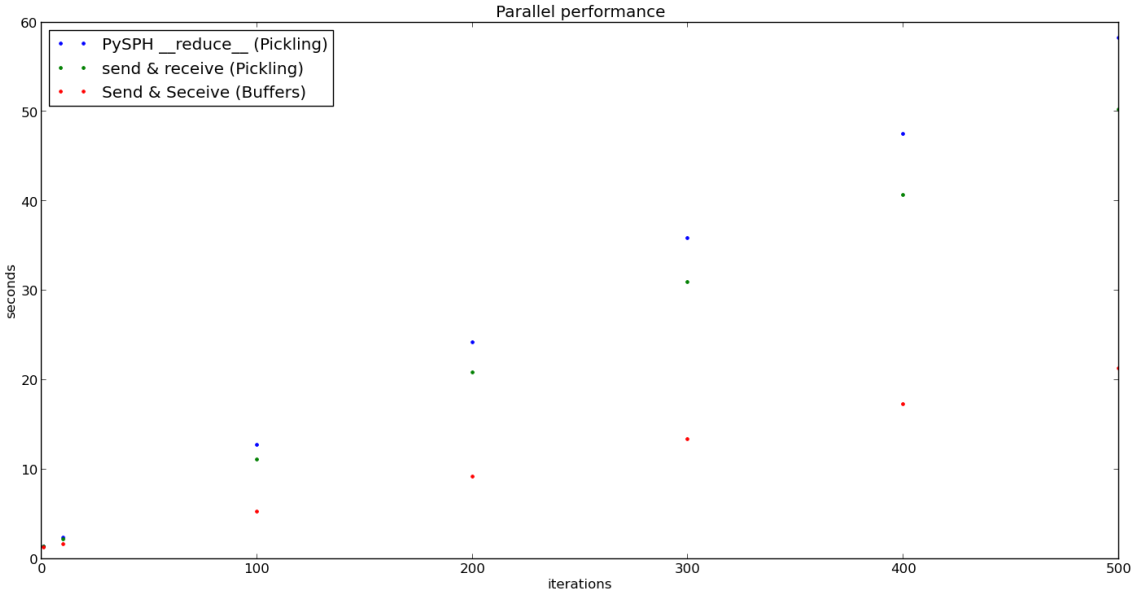

Figure 10: PySPH : Parallel performance (Multiple queries)

Thus Send & Receive method appears to be the most efficient out of all. This method may be used in the implementation of the parallelizer.

Test Send & Receive method for single and multiple queries :
This test was performed to evaluated the performance of Send & receive method in case of sending all data as a single concatenated request or as multiple fragmented requests. This test was done to decide if all particle properties should be sent in a single query or multiple fragmented queries.

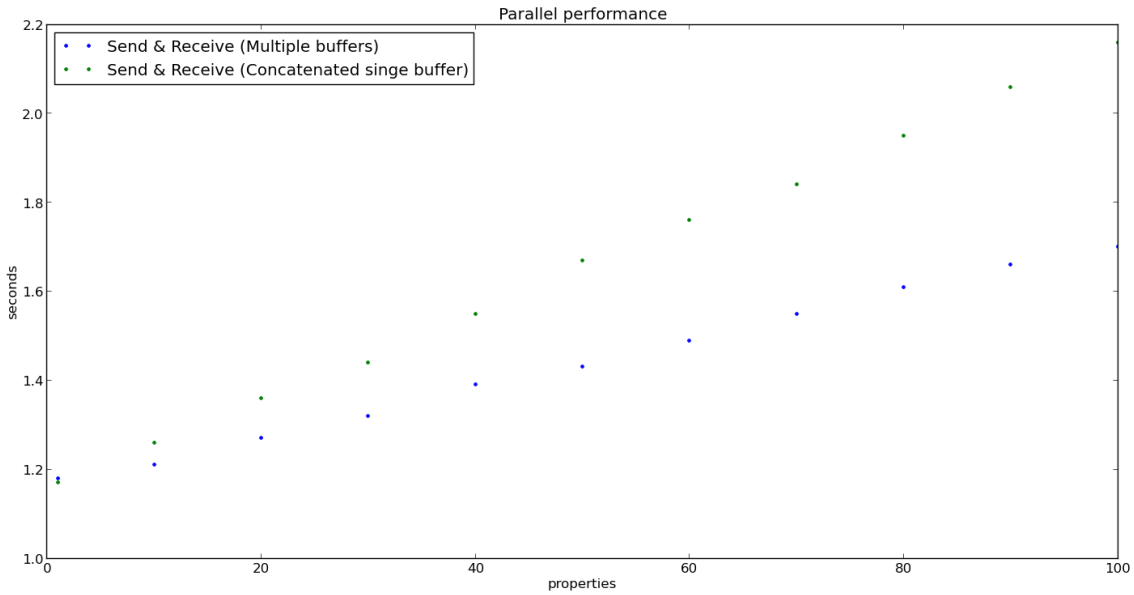

Figure 11: Send & receive : Parallel performance (Multiple vs single query)

Contrary to belief, sending multiple fragmented queries is faster than a single concatenated query. This may be attributed to the fact that a single malloc needs to be contagious in case of concatenated query and finding such a memory chunk takes more time.

It needs to be noted that all the results in this section are from code-runs on Rake (8 core Xeon system). The results may vary on machines with different configurations or across network. The results therefore need to be tested in different hardware configurations.

4 Conclusions

The requirements and tasks of the Parallelizer were finalized. Various modules which could be used in the prospective implementation were gauged. Following are certain major conclusions :

- Dynamic Load Balancing is necessary for a meaningful implementation
- Node Partitioning can not be used due to its highly expensive run-time
- Cell partitioning with a proposed cell-size of $15 \times InteractionRadius$ looks to be a good choice although testing the same code on different machines and across network is necessary
- mpi4py's Send & Receive method would be a good choice to effectuate data transfer across processors due to its efficiency and ease of implementation
- Other partitioning modules (like Zoltan) need to be explored to compare performance with Metis (which has currently been tested)

The aim of Stage II would be to use the conclusions and concepts delivered by Stage I to actually implement the parallelizer for PySPH. This parallelizer would firstly need to be faster than the serial implementation and the current parallel implementation. Also its performance and results would have to be comparable with the currently available parallel implementations of SPH.

5 References

[1] http://packages.python.org/PySPH/

[2] Chandrashekhar Kaushik, Prabhu Ramachandran; *A Python based Parallel Framework for Smooth Particle Hydrodynamics*, M.Tech Dissertation

[3] J. J. Monaghan; *Simulating free surface flows with SPH*, Journal of Computational Physics 110,399-406

[4] http://jeremybejarano.zzl.org/MPIwithPython

[5] Erik Boman, Karen Devine, Lee Ann Fisk, Robert Heaphy, Bruce Hendrickson, Vitus Leung, Courtenay Vaughan, Umit Catalyurek, Doruk Bozdag, and William Mitchell. Zoltan home page. http://www.cs.sandia.gov/Zoltan, 1999

[6] http://mathema.tician.de/software/pymetis