Parallel framework for Smooth Particle Hydrodynamics

B.Tech Project
Stage II

by

Pushkar Godbole

09D01005

under guidance of

Prof. Prabhu Ramachandran

Department of Aerospace Engineering
Indian Institute of Technology - Bombay
Mumbai

May 2013

Certificate

This is to certify that this B.Tech Project - Stage II 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 year. Working on the project has been fun and a great learning experience more so. It has instilled in me a genuine interest towards the parallel programming paradigm.

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 the design of such a parallel module and benchmark its implementation.

Keywords : Python, PySPH, Parallel computing, Load balancing, Zoltan, mpi4py

Contents

List of Figures

List of Tables

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) whose resolution 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 last version of PySPH (0.9) is capable of carrying out simulations in serial and parallel. But the 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. The report will further detail the implementation of a new parallel module that would be incorporated in the next release of PySPH.

1.2 Parallel SPH

The two primary tasks in SPH, neighbor-particle location and force computation both require an order k^2 time at best (k being the number of particles within the interaction radius). 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 neighboring processors as well. Thus an extra halo region comprising of particles near the boundaries (called ghost particles), needs to be exchanged between the neighboring 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 them-

selves in the halo region are bound to change. Hence at every time step, the halo region in each processor too needs to be updated. Additionally, as the simulation evolves in time, the initial distribution of particles across the processors is bound to get obfuscated. This would highly increase the number of ghost particles. Hence a periodic load-balancing step has to be executed to appropriately re-distribute the particles across processors.
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 has to be built upon the PySPH framework. The implementation thus needs to facilitate switching from serial to parallel execution without the end-user having to tweak the problem description too much. It must be as de-linked from the actual PySPH solver as possible. This chapter will detail the anatomy of current PySPH in brief. Although this anatomy is about to change to a certain extent. These prospective changes have been described in a later chapter.
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.

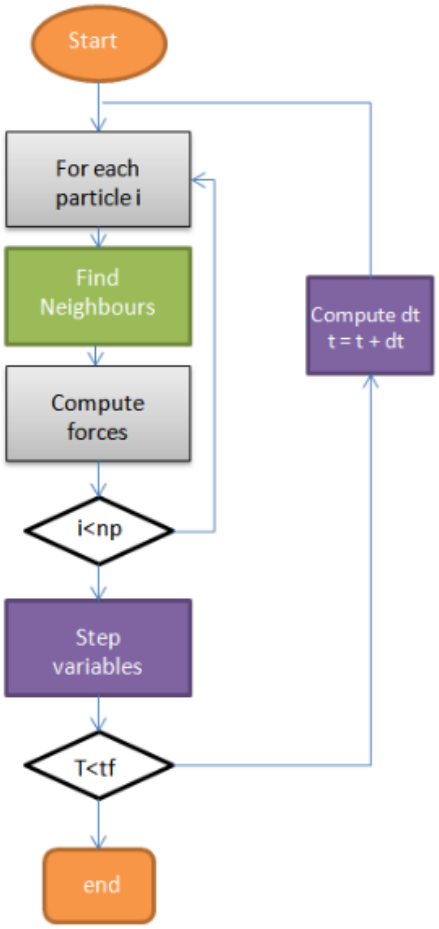


Figure 1: PySPH flow-chart

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.

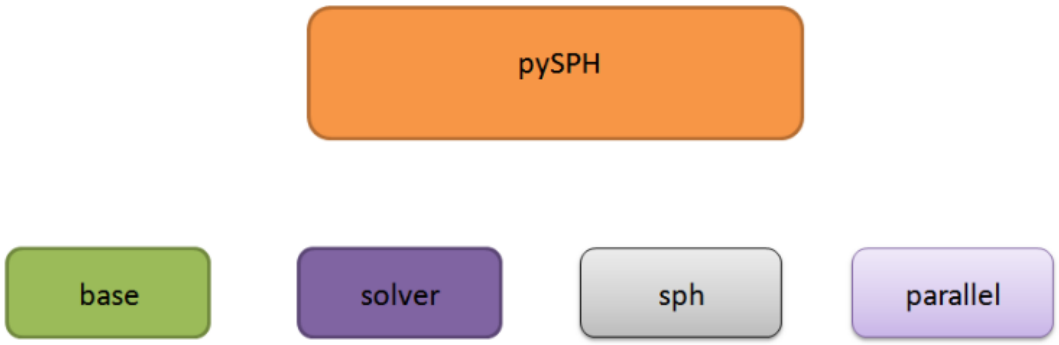


Figure 2: PySPH modules

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 neighbors 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 module's 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.
The parallelizer therefore has three primary tasks to accomplish efficiently :

- **Load-balancing :** Achieve and maintain a balanced workload (in terms of local particles) across processors
- **Halo-minimization :** Minimize the the halo region (number of ghost particles) that needs to be exchanged between neighboring processors at every time-step
- **Data-transfer :** Achieve prompt exchange of ghost-particles at every time step and export-particles after every load-balancing step

It must be noted that, the ghost particles from the neighboring 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 neighboring processors.

2.4.1 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 size of the halo region would depend upon three factors :

- **Thickness of the halo-region :** Determined by the interaction radius.
- **Connectedness of the partitions :** Volume of the halo region would be minimized by maximizing its connectedness
- **Surface area of the partition :** Minimizing surface area, the direct determinant of the common region that needs to be shared between processors; would minimize the halo region

2.4.2 Load-balancing

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.

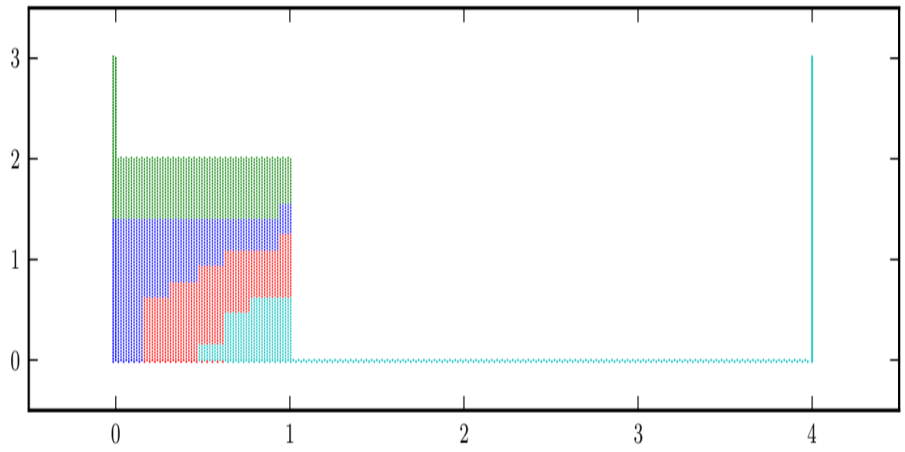


Figure 3: Dam break : Initial partitioning

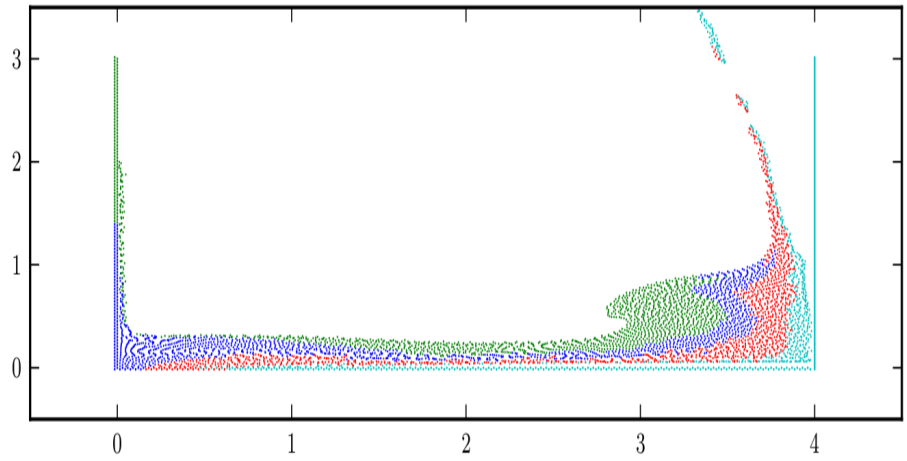


Figure 4: Dam break : Obfuscated partitioning due to problem evolution

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.

2.4.3 Data transfer

The data transfer would occur at two stages of the simulation.

- **Ghost particles :** After the ghost particles are determined, they need to be sent across to their appropriate destination processors at every time-step
- **Export particles :** After every periodic load-balancing step, the processors would possess a list of particles that need to be exported to neighboring processors

The data transfer needs to be efficient enough to speedily execute the above two tasks without becoming a bottle-neck in the entire parallelization process.

3 Stage I Overview

The objective of stage I was to define the requirements of the parallel module, evaluate methodologies for an efficient and valid implementation and conceptually design the parallelizer. Stage I delivered two important conclusions.

3.1 Node partitioning vs Cell partitioning

The load-balancing may be done either using particles or cells (containing particles) as entities. The cell-partitioning essentially has lesser entities and coarser connectivity. This makes the partitioning more efficient w.r.t. time and less efficient w.r.t. load-balancing. The graph partitioning module 'Metis' was used to evaluate the two approaches.
n particles were distributed randomly in a 10x10 square. PyMetis was used to partition the problem space into 10 chunks.
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.

Partitioning time : 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).

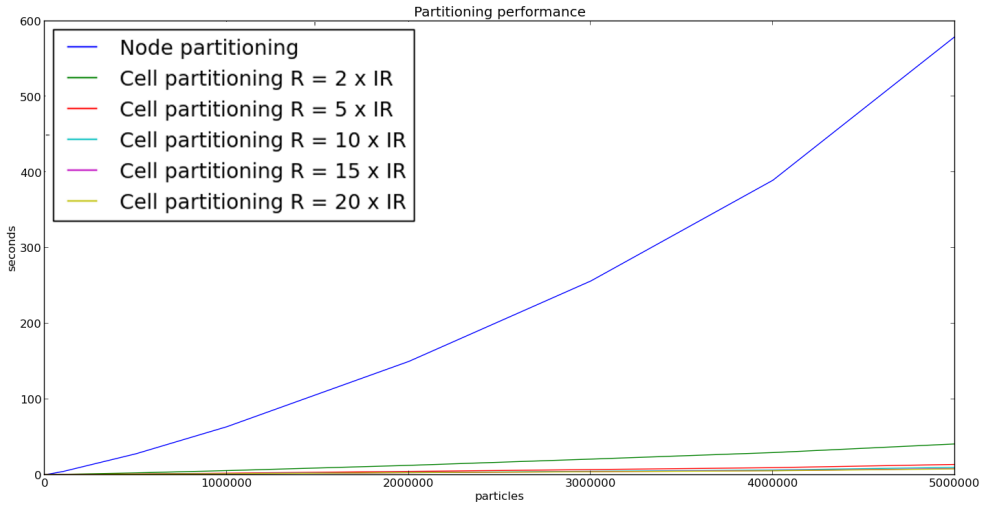


Figure 5: Partitioning : Time performance

It can be seen that Node-partitioning is taking extremely long compared to Cell-partitioning. 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.
It needs to be noted that increasing cell-size has diminishing returns in terms of time

advantage.

Partitioning quality : 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)*. Root mean square deviation of the particle distribution across all chunks is plotted in the graph below for various cell-sizes in terms of interaction radius for 1M particles.

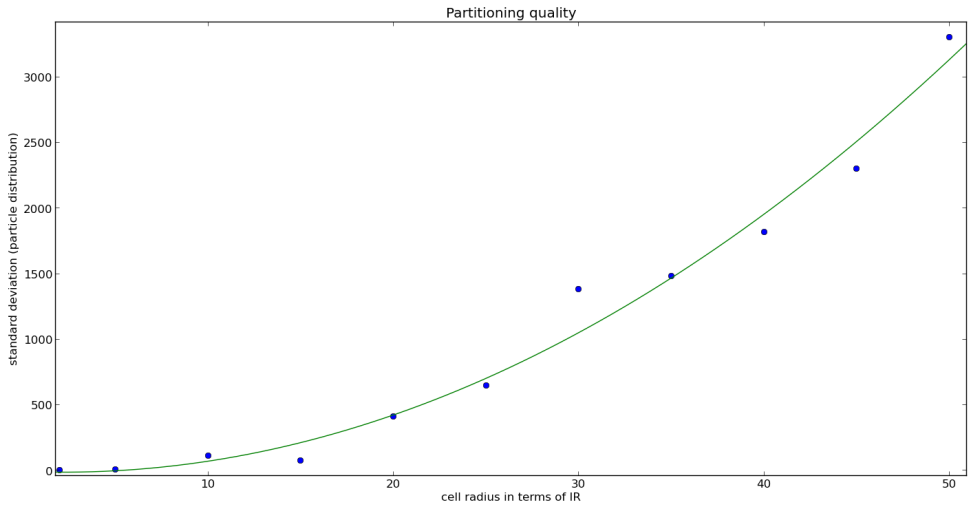


Figure 6: Cell partitioning : Quality performance

It can be seen that the deviation increases quadratically with cell size. At cell radius of $50 \times IR$ the deviation has crossed 3000. This means that some chunk might have 3000 particles more/less than the mean and hence is handling an unbalanced load.

3.2 Data transfer

With Load-balancing done and ghost particles determined, the next task is to actually transfer the particle data across processors. **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.

3.2.1 mpi4py Methods

Two message passing techniques implemented in MPI :

- **Point-to-point communication :** One sender one receiver communication
- **Collective communication :** One-to-many and many-to-one communication

Two message passing protocols to transfer the data :

- **Pickling :** Encodes data to be sent using Python's **Pickling** module. Pickle is then sent across the processors. 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). Not an efficient means to transfer data

- **Buffers :** Does away with the encoding and decoding process and directly sends the raw data across as Python's numpy arrays. The receiver is required to know the size of the data it intends to receive from the sender before hand. Due to elimination of pickling, the process is highly efficient

Various requirement specific data transfer methods exist in mpi4py. These methods fall in one of the two categories each mentioned above.

- **send & receive :** Point-to-point pickling
- **Send & Receive :** Point-to-point buffers
- **Scatterv :** One-to-many buffers. Specifically suitable for codes where the root node remains fixed. When implemented for variable root-node case, the run crashes due to requirement and lack of synchronicity
- **Gatherv :** Many-to-one buffers. Primary difference from Scatterv is that every process depends on itself. Hence suitable for variable roots too. The defining of the receive array and gathering of data occur on the same process. Eliminates the need of synchronicity

3.2.2 Performance analysis

Data transfer across processors using mpi4py on Rake (8 core Xeon processor) was gauged with the following test case :
Initiate 4 parallel processes using mpi4py. Each process sends its own data to other 3 processes.

Data-transfer with increasing number of particles :
Performance wise Send & Receive, Scatterv and Gatherv all perform equally well. send & receive was found to be extremely slow as anticipated initially due to pickling. Gatherv was eliminated due to its synchronicity issue.

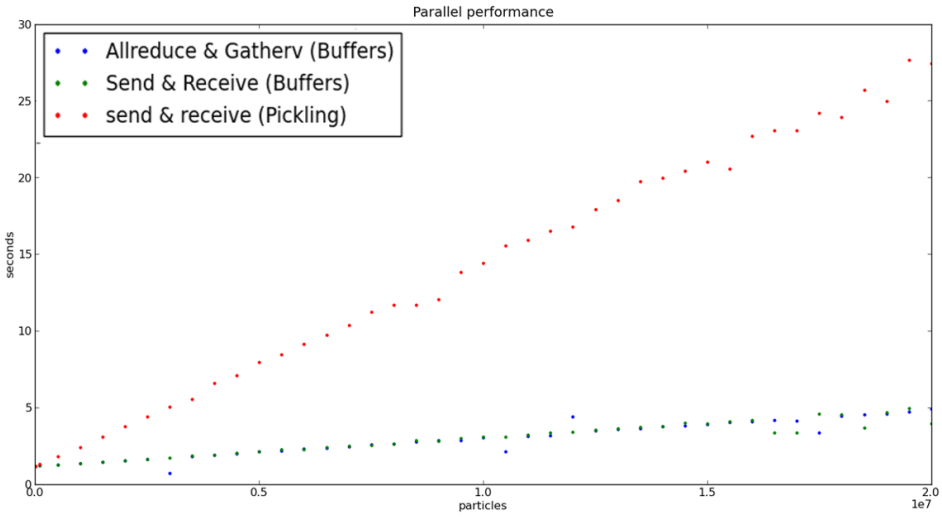


Figure 7: mpi4py : Parallel performance

Since no significant difference between Send & Receive and Scatterv was found in terms of time-efficiency, Send & Receive was finalized as the protocol due to its ease of implementation.

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.

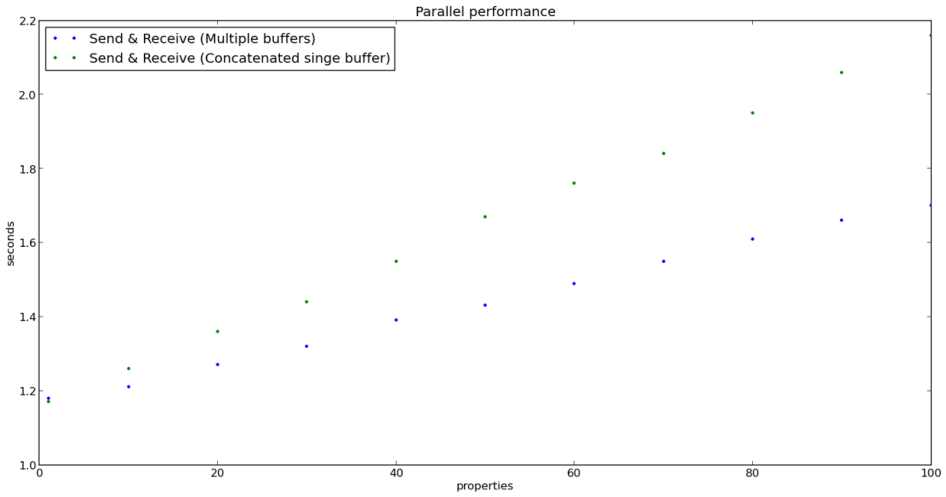


Figure 8: Send & receive : Parallel performance (Multiple vs single query)

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.

3.3 Conclusions

- Dynamic Load Balancing is necessary for a meaningful implementation
- Node Partitioning can not be used in the long run due to its highly expensive runtime
- mpi4py's Send & Receive method would be a good choice to effectuate data transfer of arrays across processors due to its efficiency and ease of implementation
- AllReduce may be used for determining sizes of arrays to be received from neighboring processors
- Parallel partitioners like Zoltan, ParaMetis need to be explored for executing the load-balancing in parallel unlike Metis

4 Parallelizer design

The parallelizer has three major tasks to accomplish :

- Effectuate load-balancing across all processors and generate list of export particles
- Identify the remote cells for every processor and generate a list of ghost particles
- Execute data-transfer of the export particles after load-balancing and ghost-particles after remote-cell identification

The current implementation of the parallelizer uses the Zoltan library to effect load-balancing and remote-cell identification. A python wrapper for the same (PyZoltan) has been written by Kunal Puri. The parallelizer implements the two tasks via this wrapper.

4.1 Load-balancing

Load balancing requires the particle distribution information as input in order to partition the data. Although cell partitioning has been proved to be more efficient and effectual for a multitude of reasons (mentioned later in the report), the current working implementation of the parallelizer takes particles as the partitioning objects. The Zoltan library has various types of partitioners which may be sub-categorized as :

- Geometric partitioners
- Graph partitioners
- Tree partitioners

Out of these the current parallelizer makes use of the geometric partitioner techniques to execute the load-balancing. These algorithms require as input, the coordinates of objects to be partitioned. For a particle method like SPH, the particle positions serve as the most natural input.

Graph partitioners are generally suitable for data having inherent connectivity information. Particle data also may be re-interpret as graphs but morphing the particles as nodes and nearest neighbors as the connected particles via implicit edges. But as seen in the cell vs node partitioning, this is highly inefficient in terms of time required as the implicit graph becomes highly connected due to large number of neighbor particles.
Graph partitioning may be implemented in the parallelizer only by porting to cell based partitioning. In this case each object has at most 8 neighbors in the 2D case and 26 in the 3D case. Migration to cell based scheme is under progress after which the partitioning may be extended to graph techniques too.

In the implementation of the parallelizer, the load-balancing is effectuated as follows :

- Each processor creates its own set of particles initially
- Load balancing is subsequently executed in every time-step to distribute particles across processors. (This is unavoidable in the current implementation)
- Each particle is assigned a local id (lid) : serial no. inside the processor; and a global id (gid) : a global serial no. unique to each particle
- This is required by the load-balancing function (Zoltan_LB_Balance in this case) which executes the load-balancing

- The load-balancing function takes the gids, lids and particle data (co-ordinates in case of Geometric partitioners)
- It returns a list of export particles and export processors (destination processor) corresponding to the export particles and similarly a list of import particles and import arrays (which is redundant)
- Using the two lists, export data of every processor is transmitted to appropriate destinations using the designed data-transfer protocol

All processors have their new set of particles at the end of load-balancing. These particles now need to be re-binned into cells before proceeding to the ghost computation.

4.2 Ghost computation

The primary bottle-neck in parallelizing CFD solvers in general and SPH in particular is that, the problem can not be split into independent sub-problems. The chunks depend on each other for proceeding to the next time-step. Thus the particle data at the boundaries has to be exchanged between neighboring processors. These boundary particles, ghost-particles as they are called, have a special characteristic. The ghost-particles influence the real particles but are not (or rather, need not be) influenced by the real particles. Hence the name.
Thus, before proceeding to the solver, the boundary data has to be conveyed to all the processors. The process is implemented as follows :

- For every cell a bounding box size (2 x IR) more in every direction along every dimension is constructed. (This is equivalent to a cell of size 9X of the current cell size in case of 2D.)
- Since influence of any particle is limited to only (2 x IR), all the processors that have some portion in common with this bounding box will be the neighboring processors of this cell
- Currently the Zoltan_PP_Box_Assign function is used to identify the overlapping processors. The function returns a list of processors that overlap this bounding box
- If the returned list contains only the parent processor of a cell, the cell is not a boundary cell and may be ignored in the ghost computation
- Note that, due to the use of Zoltan's inbuilt function, update of Zoltan's cell map becomes inevitable at every time step.
- Easiest way to do this is to run the Zoltan_LB_Balance function at every time step
- Thus using Zoltan_PP_Box_Assign has constrained us to execute the load-balancing at every time-step
- Once the overlapping processors are identified, all particles in that cell are to be sent to the appropriate processor item An array of such particles is created and sent to the neighbors. These particles get appended at the end of the neighbor processor's particle array and are destroyed after the solver step completes.
- This procedure is repeated at every times step.

4.3 Data transfer

As already mentioned, the data transfer occurs at two levels in the simulation. After load-balancing and after ghost computation. The data transfer is occurring simultaneously across multiple processors. The protocol for data transfer therefore had to be critically designed to avoid an deadlocks and losses. Following is the protocol followed to ensure a hassle free data transfer :

- Note that the Send & Receive method used is a blocking data-transfer technique
- Function takes list of Gids/Lids and corresponding destination processor array along with particle data as input
- First all destination processors are notified about the size of the data to be expected by the source processors (Each processor is a source as well as a destination in our case)
- The destination processors define their receive buffers from this size information
- All processors instantiate a Send query to all destination processors with a lower rank than self
- All processors instantiate Receive query to receive from all the sending processors
- All processors now receive data from source processors with higher ranks
- All processors instantiate Send query to all destination processors with higher rank than self
- The remaining Receive queries now get completed by receiving data from processors with lower rank
- Non-blocking communication (Isend & Ireceive) was attempted with unique tags. But the process crashed and data was lost
- Hence this 3-step protocol with blocking Send & Receive requests is effectually implemented

4.4 Zoltan

Zoltan is a C library developed by the Sandia National Laboratory for parallel partitioning algorithms and data migration tools. Zoltan employs various parallel combinatorial algorithms for unstructured and/or adaptive computations. Zoltan has support for Geometric, Graph and Tree based partitioners in parallel. The Geometric partitioners from Zoltan :

1. Recursive Coordinate Bisection (RCB)
2. Recursive Inertial Bisection (RIB)
3. Hilbert Space Filling Curves (HSFC)

have been implemented in the parallelizer.
Zoltan's graph-partitioning techniques are most suited for mesh based schemes. The graph-partitioning techniques could be implemented in the parallelizer after migrating to cell partitioning with the cell data re-interpreted as mesh data. Besides, Zoltan also provides support for other popular graph partitioners like Metis, ParMETIS and Scotch.

The current PyZoltan wrapper supports only geometric partitioning. Wrapping of the graph partitioning functions is under way.

Zoltan's two important functions Zoltan_LB_Balance (for load-balancing) and Zoltan_PP_Box_Assign (for ghost particle location) form the cruz of the parallelizer currently. Migrating away from these functions to make the parallel module independent of Zoltan is under way.

4.5 Periodic Load-balancing

Current implementation makes it inevitable to execute load-balancing at every time-step. This is primarily due to the dependence of ghost computation on Zoltan. Ideally, load-balancing should occur only when the particle distribution gets highly obfuscated leading to unbalanced work load in terms of particles per-processor or steep rise in the halo-region. This approach is called Periodic load-balancing and needs to be implemented in the parallelizer.

The parallelizer would keep track of the particles held by each processor and the amount of ghost-particles exchanged in every iteration. Load-balancing step would be executed only after certain threshold imbalance or data-transfer exceeds.
The current ghost-computation works error free as load-balancing step is executed at every time-step. Hence the cell data is automatically updated in Zoltan and Zoltan_PP_Box_Assign returns the correct set of remote particles. With periodic load-balancing, the cell distribution data will not be updated at every iteration. Hence any new cells formed or old cells destroyed will not be detected by the Zoltan_PP_Box_Assign function in between two load-balancing steps. It is also not suitably possible to just update the cell distribution data in Zoltan as Zoltan has internalized the entire data-structure and at least a python wrapper can not modify this data.

The only viable option that remains is to implement an in-house ghost computation function. The implementation of this function may be rendered as follows (For 2D case) :

- Iterate over all cells to check if each has 8 neighbors. Mark cells with less than 8 neighbors as "boundary cells"
- Look for presence of either of the 9 neighbor cells of every boundary cell in every other processor : 9 x O(1) operation per boundary cell (hash lookup)
- If found, all particles in that boundary cell are ghost-particles for that neighbor processor
- Create list of such particles and the destination processors
- Use the data transfer protocol to effectuate the necessary transfer

Thus the parallel module was designed and implemented.

- Cell based partitioning
- Graph techniques for partitioning
- Periodic load-balancing

are the three major features that remain to be added. Besides an in-house parallel load-balancer by the name "K-means" is being developed by Kunal Puri. This would be distributed as the default load-balancer in the next release version of PySPH while Zoltan would be an optional dependency.

5 Load-balancing techniques

Zoltan has a multitude of load-balancing techniques. The parallel module currently supports the geometric techniques and would soon support Graph based methods too (after cell-partitioning is implemented).

5.1 Geometric partitioners

These algorithms require as input, the coordinates of objects to be partitioned. This is the most suitable and easy to implement partitioning technique for SPH like particle methods. Here we go through the geometric partitioners that have been implemented (via Zoltan) in the parallelizer.

5.1.1 Recursive Co-ordinate Bisection (RCB)

The RCB algorithm is the simplest of the geometric partitioners. It begins bisecting the domain using a plane orthogonal to the coordinate axes. This generates two sub-regions with equal number of particles. This algorithm is then applied recursively to each sub-region to reach the desired number of partitions. Weights can be assigned to particles to achieve different load balancing criteria. In the current particle based partitioning, no weights are assigned to the nodes. But after migrating to cell-partitioning the cells would be assigned weights equal to the number of particles held by the cell for equitable distribution of the particles across processors. The principle advantage of the RCB method is speed and simplicity. RCB is not suitable for arbitrary domains. The partitioning generated by RCB is always orthogonal to the standard axes irrespective of the particle distribution. This leads to unnecessary rise in the halo-region as may be seen in the example later.

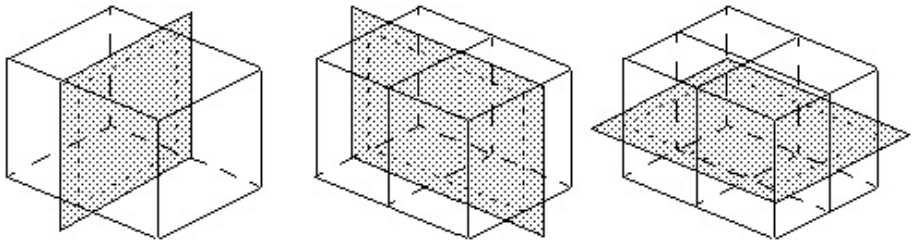


Figure 9: RCB partitioning

5.1.2 Recursive Inertial Bisection (RIB)

RIB is variant of the RCB method that works well for non axis aligned domains is the recursive inertial bisection method. In this method, the bisection line is orthogonal to the principle inertial axis. The method produces visibly better partitions for arbitrary domains like in the dam-break problem.

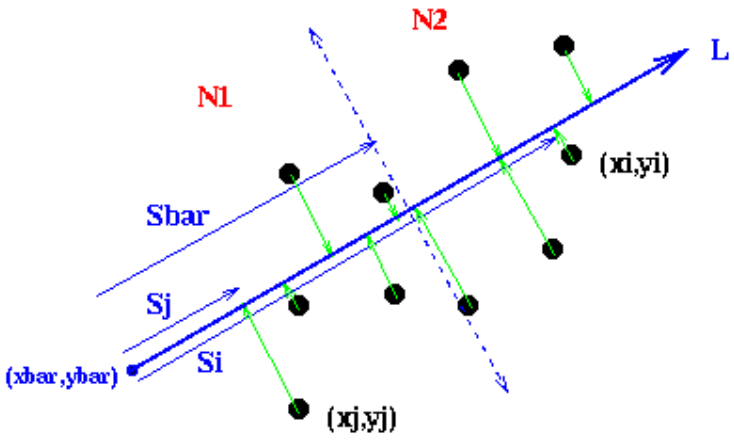


Figure 10: RIB partitioning

5.1.3 Hilbert Space Filling Curves (HSFC)

Hilbert Space Filling Curves are functions that map a given point in a domain into the unit interval $[0, 1]$. The inverse functions map the unit interval to the spatial coordinates. The HSFC algorithm divides the unit interval into P intervals each containing objects associated to these intervals by their inverse coordinates. N bins are created to partition the interval and the weights (number of particles in our case) in each bin are summed across all processors. The algorithm then sums the bins and places a cut when the desired weight for a partition is achieved. The process can be repeated to improve the partitioning tolerance. The algorithm basically maps a 2D\3D domain into 1D line which twists at every half. The tolerance of the method improves as the value of N increases.

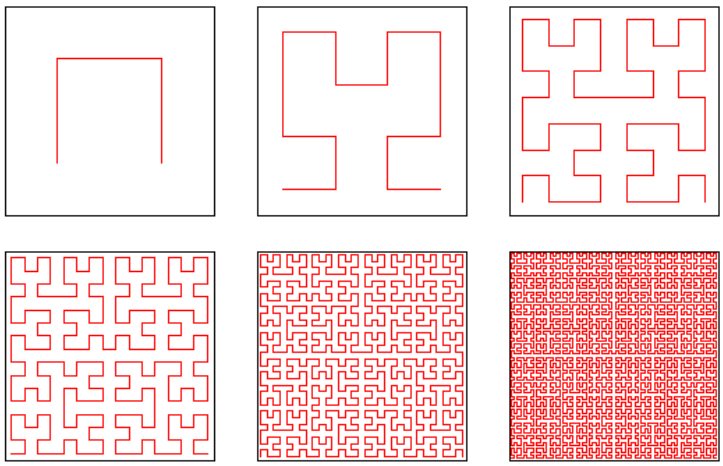


Figure 11: HSFC partitioning

5.2 Graph partitioners

Graph partitioning techniques used for mesh based schemes re-interpret the 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. In SPH, the particles could be treated as the vertices in the graph while the nearest neighbors could define the edges emanating from a given vertex. While this works in principle, the resulting graph has too many edges, leading to 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 neighbors (26 in 3D and 8 in 2D) define the node connectivity. The graph is now invariant to the number of neighbors for a given particle (resolution in SPH). Nodes can be weighted based on the number of particles in each cell to achieve a good load balance across processors. The cells however if duplicated across processors (i.e. 2 cells in 2 different processors with the same physical location) must have unique global ids. As the cells remain stationary, their neighbors need not be located and would already be known before hand, like in case of meshes. The advantage of graph partitioners is that they produce fast and high quality partitions.

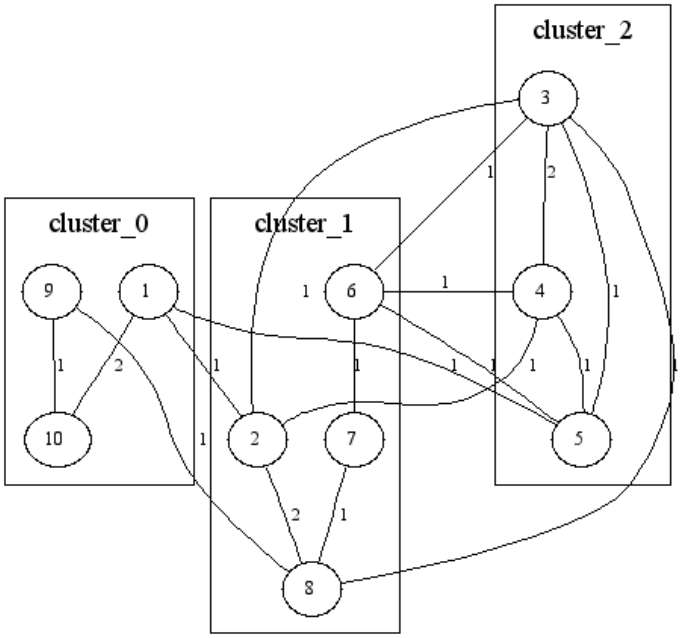


Figure 12: Graph partitioning

5.3 Load-balance comparison

Zoltan also has support for Tree based partitioners, but their applicability in SPH based scenarios is rather limited. Besides Zoltan's Graph partitioner, Metis's serial graph partitioner was also explored. The partitioning visualization in the following figures clarifies that RIB is the most suitable method for arbitrary domains (very common to SPH)

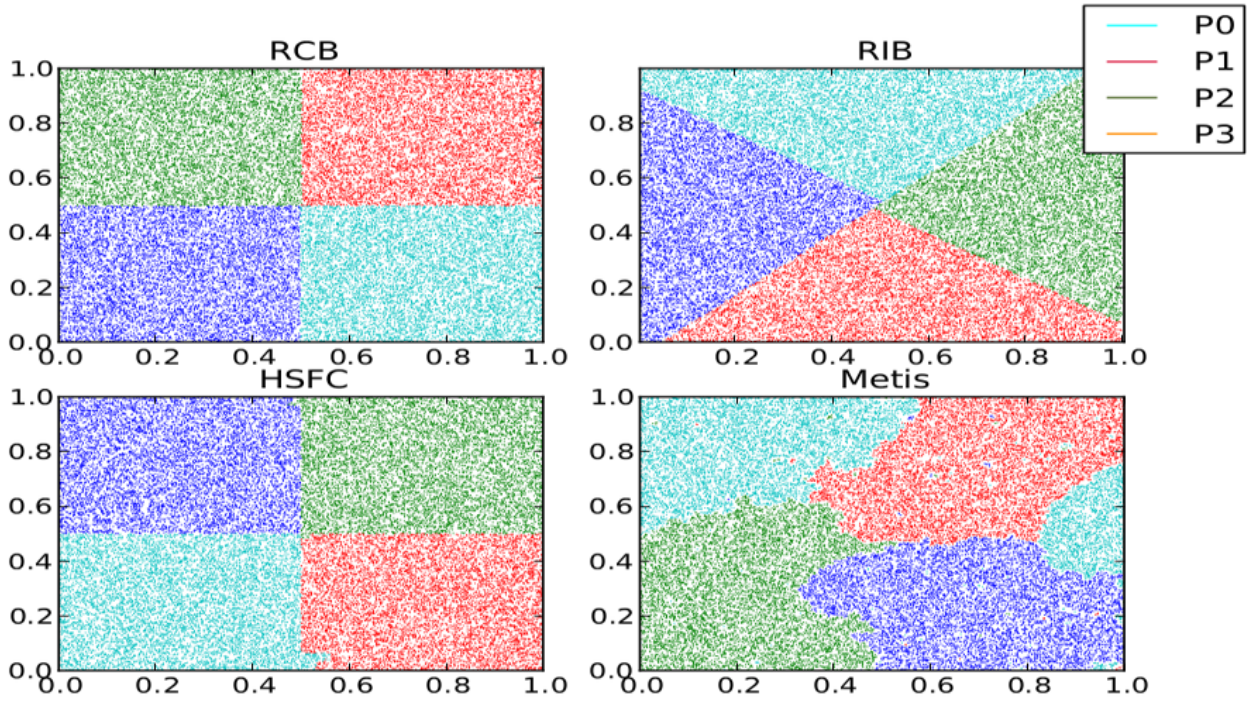


Figure 13: Block distribution : 4 partitions

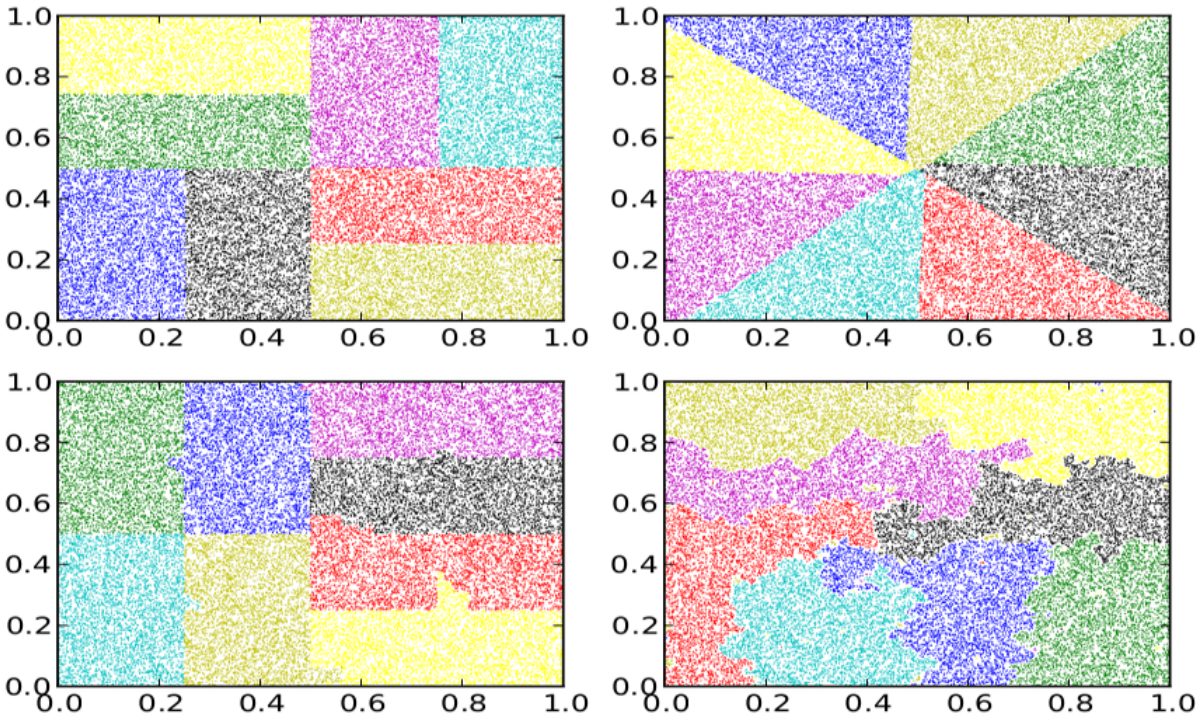


Figure 14: Block distribution : 8 partitions

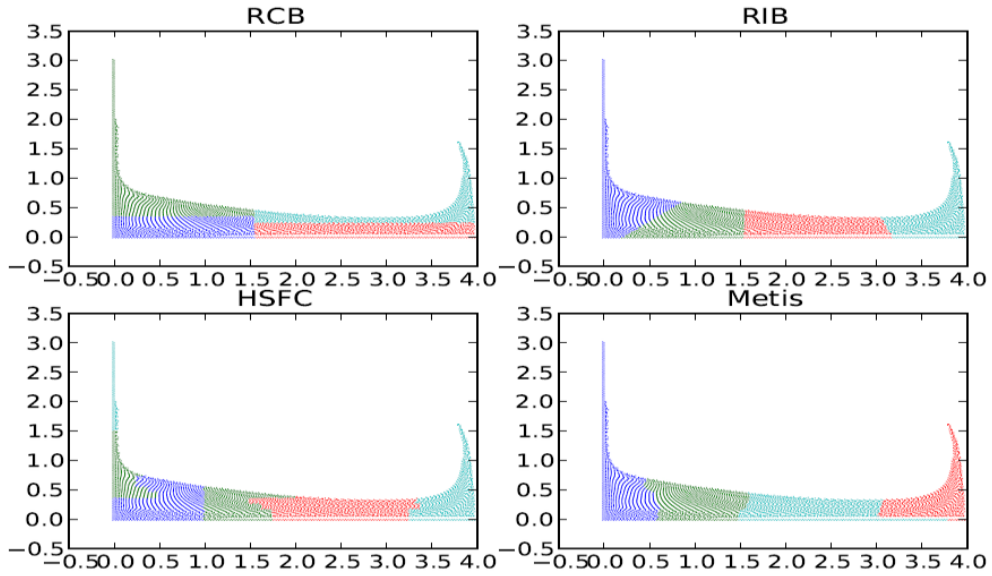


Figure 15: Dam-break distribution : 4 partitions

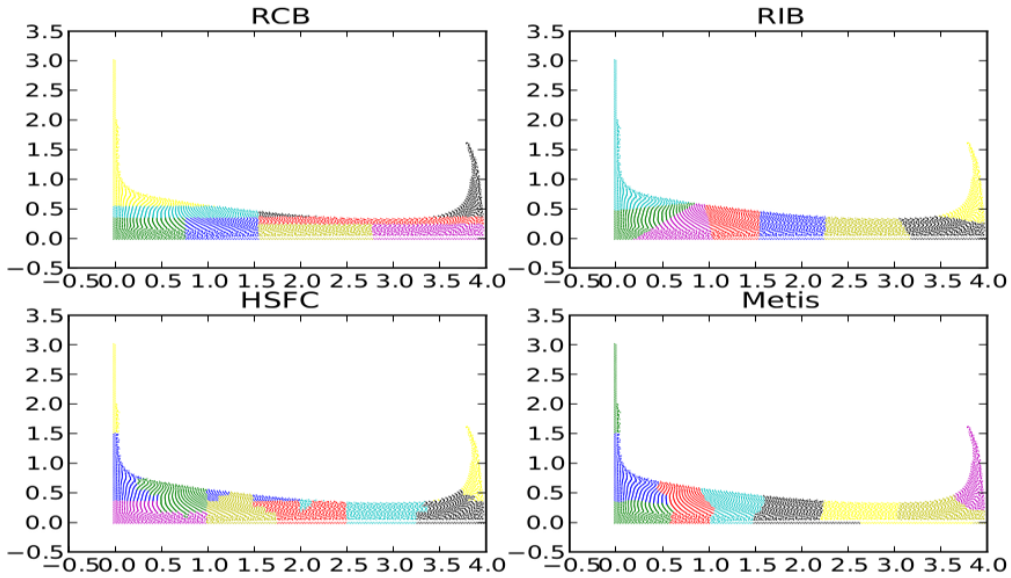


Figure 16: Dam-break distribution : 8 partitions

6 Symmetric interactions

The two primary functions in SPH are, neighbor particle location and acceleration computation. Major portion of the simulation time is dedicated to these tasks. Symmetric interactions is a technique that exploits the inherent symmetry in SPH to speedup the process.
While computing the acceleration of a particle A in a time-step, first the neighbor particles of A are identified. The contribution from each of the neighbors is added and the cumulative acceleration at A is calculated. If A is a neighbor of B, B has to be a neighbor of A and hence by Newton's third law, the influence by A on B is the same as the influence by B on A. These two facts are made use of to halve he number of computations in the simulation. Symmetric interaction was implemented using two methods as described below :

6.1 Gid/Lid based Symmetric Interactions

- For every particle in the processor, all the neighbor particles are located
- List of neighbor particles used to compute accelerations
- Acceleration contribution only from particles with higher gid/lid than sel is considered and added to the acceleration term of **both** particles
- Thus only part of the neighbors located by the neighbor locator are considered
- Total number of acceleration computations halved
- Speed-up of 16% was observed over the older version of zsph

6.2 Cellwise Symmetric Interactions

- In symmetric interactions, once a particle's influence is considered on all neighbors, it may be ignored to the extent of assuming that it does not exist
- A flag is added as an attribute to every cell. Flag marked as True once all the particles' accelerations are computed in that cell
- Neighbor particle locator only looks into cells with a False flag
- For these cells, symmetric interactions is applied for every particle contained in the cell. Finally the cell is marked as done (True)
- This method further optimizes the process by not doing neighbor computation on the cells whose contribution has already been considered
- Speed-up of 10% was observed above Gid/Lid based symmetric interactions

Thus symmetric interactions is implemented in the new PySPH.

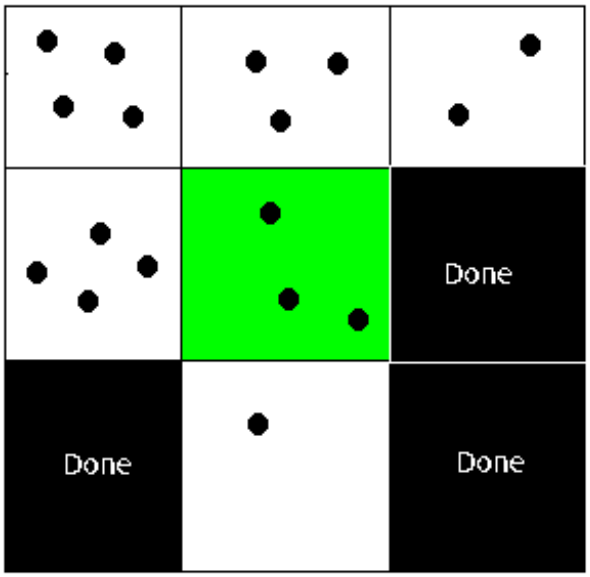


Figure 17: Cellwise symmetric interactions

7 Results

The results have been generated on machines in the CFD lab. Here is a table containing the basic information about the machines used.

Most codes were run on machines marked green. Codes were also run on CDAC's Param

Machine	**IP**	**Processor**	**RAM**
Rake	10.101.11.65	Intel Xeon 8 core	15.7Gb
Vorton	10.101.11.66	Intel Xeon 8 core	15.7Gb
Dorado	10.101.11.107	Intel Xeon 16 core	15.7Gb
Orion	10.101.1.13	Intel Xeon 8 core	31.5Gb
Scorpio	10.101.22.7	Intel i5 4 core	3.7Gb
Aquila	10.101.11.101	Intel i3 4 core	3.7Gb
Draco	10.101.1.15	Intel i3 4 core	3.7Gb
Gemini	10.101.11.103	Intel i3 4 core	3.7Gb
Nebula	10.101.2.15	AMD 18 x 12 cluster	18 x 12 Gb

Table 1: CFD lab : Machines

system remotely. But the system was found to be excruciatingly slow compared to the in-house machines. Dam break problems took 5-6 time more time to run on Param compared to Rake/Vorton.

7.1 Network characterization

Before running any codes on the machines, they were characterized using standard benchmark tests. Machines were characterized based on internal and network Latency, Bandwidth and computation power. HPC Challenge Benchmark tests' STREAM and b_eff were used to characterize the computation power and network respectively.

7.1.1 Latency & Bandwidth

Latency is a measure of time delay experienced in a system, the precise definition of which depends on the system and the time being measured. In communications, the lower limit of latency is determined by the medium being used for communications.
Bandwidth is a measurement of bit-rate of available or consumed data communication resources expressed in bits per second or multiples of it

The b_eff test measures latency and bandwidth for three topologies as follows :

- **Ring :** Processors arranged in a fixed order in the ring throughout the test
- **Ring & Random :** Processors arranged randomly in the ring at every pass throughout the test
- **Ping-pong :** Non-simultaneous send-receives between all possible processor pairs

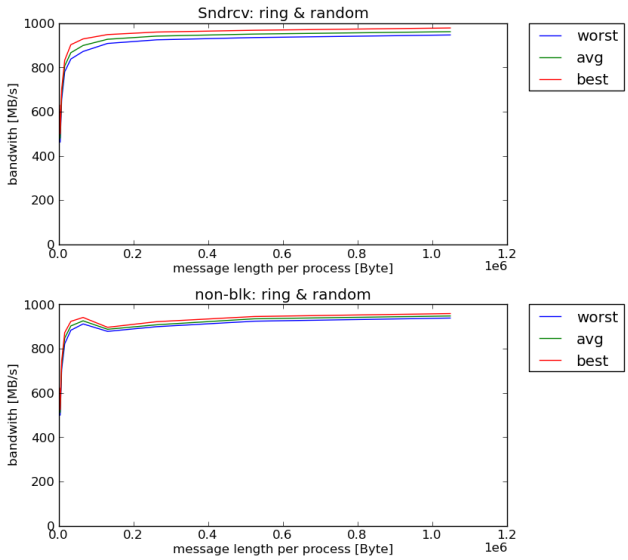
Latency :

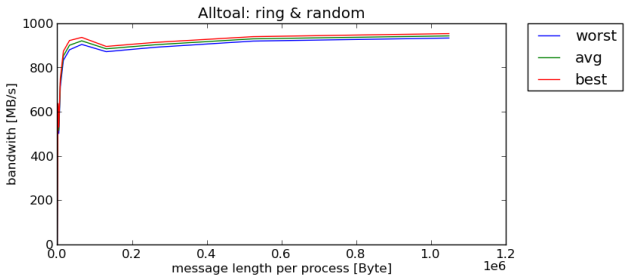
Machine \Parameter	**Latency rings (microsec)**	**Latency Rings & random (microsec)**	**Latency ping-pong (microsec)**
Gemini (4 procs)	1.752	1.748	0.879
Draco (4 procs)	1.522	1.529	0.73
Gemini-Draco (4+4 procs)	36.536	73.176	1.049
Gemini-Draco (2+2 procs)	68.839	103.426	0.596

Table 2: Latency characterization

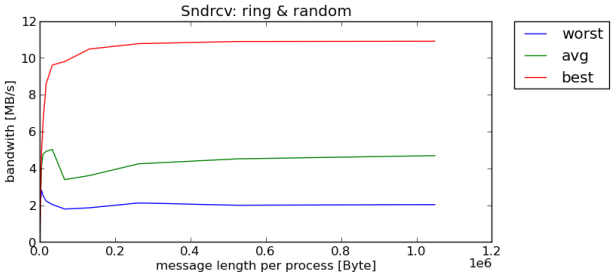
These b_eff tests crashed on Rake, Vorton and Nebula. Hence the latency and bandwidth of and across these machines could not be characterized.

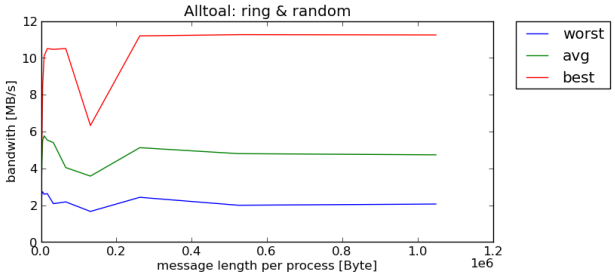
Bandwidth :

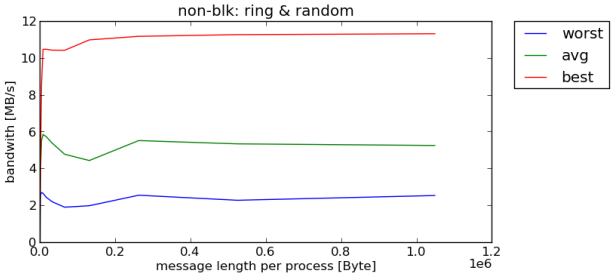




Gemini Bandwidth







Draco-Gemini Bandwidth

7.1.2 Computation rate

Gemini :
Array size = 150M (elements)
Memory per array = 1144.4 MiB (1.1 GiB)
Total memory required = 3433.2 MiB (3.4 GiB)

Function	**Best Rate (MB/s)**	**Avg time (sec)**	**Min time (sec)**	**Max time (sec)**
Copy	8600	0.283036	0.279071	0.298111
Scale	8546.6	0.284758	0.280814	0.297675
Add	9128	0.396136	0.394393	0.402997
Triad	9115.3	0.396303	0.394939	0.399162

Table 3: Gemini : Computation rate

Vorton :
Array size = 300M (elements)
Memory per array = 2288.8 MiB (2.2 GiB)
Total memory required = 6866.5 MiB (6.7 GiB)

Function	**Best Rate (MB/s)**	**Avg time (sec)**	**Min time (sec)**	**Max time (sec)**
Copy	3479.9	1.386182	1.379369	1.392447
Scale	3488	1.387069	1.376131	1.394406
Add	4533.2	1.594228	1.588294	1.606344
Triad	4528.6	1.593789	1.589878	1.605193

Table 4: Vorton : Computation rate

7.2 Load-balancing comparison

Problem : dam_break_obstacle with 70000 particles on 8 processors of Rake (Simulation time = 3 sec)

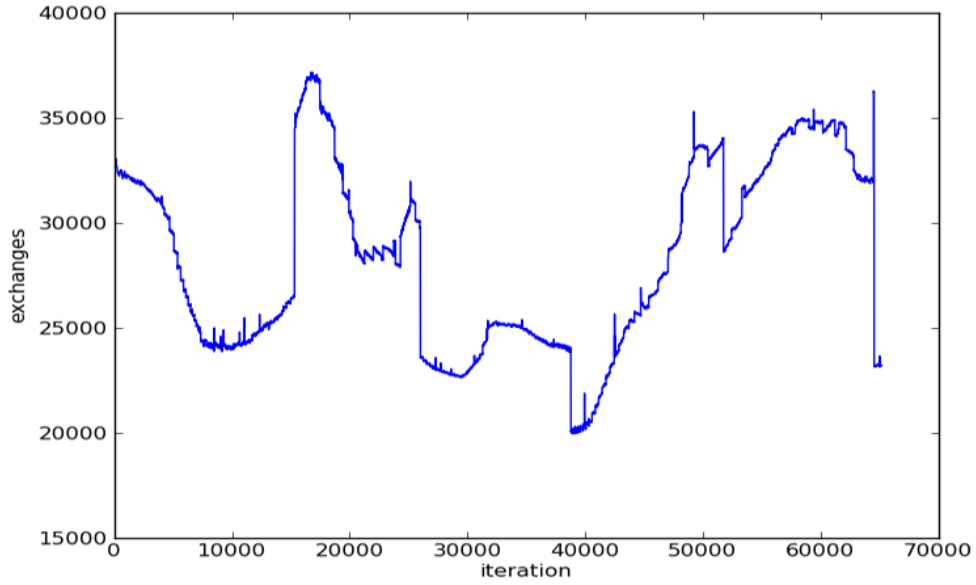


Figure 18: RCB : particles exchanged

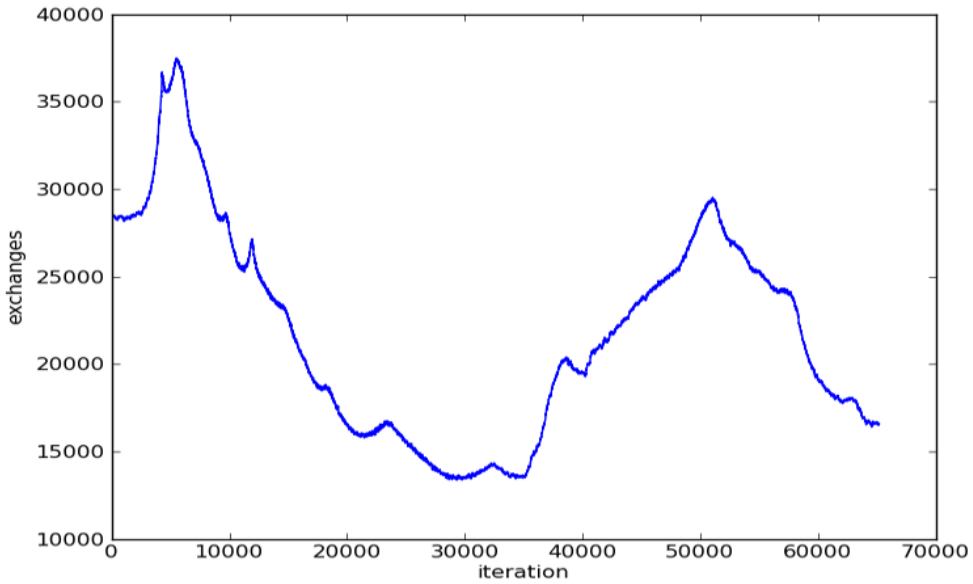


Figure 19: RIB : particles exchanged

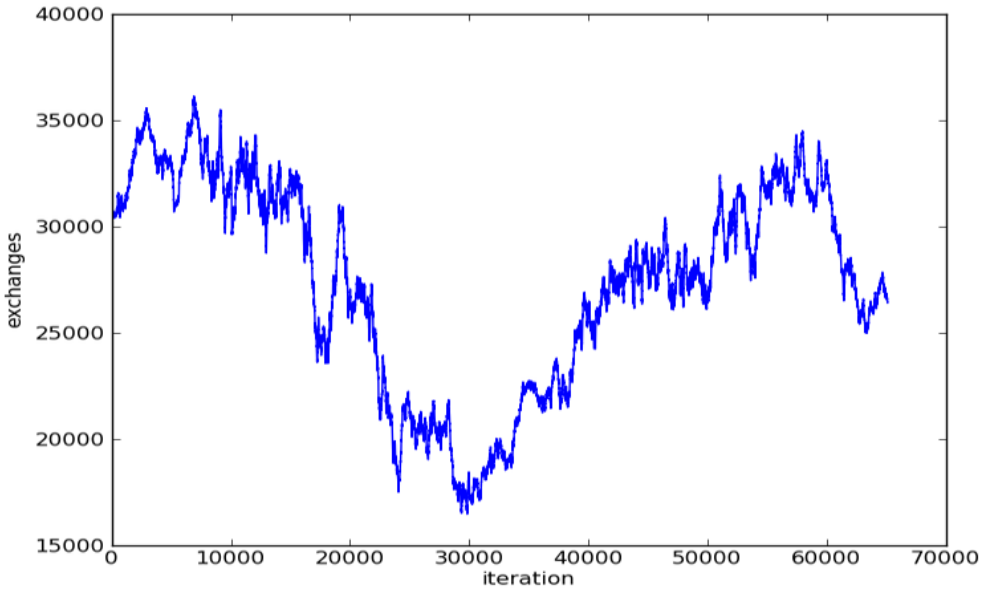


Figure 20: HSFC : particles exchanged

Method	**Time**	**Total exchanged** textbfparticles
RIB	05:00:01	1.42 x 10^9
RCB	05:05:41	1.84 x 10^9
HSFC	05:09:58	1.78 x 10^9

Table 5: Load-balancing data-exchange comparison

Here it can be seen that the total number of particles exchanged over the entire simulation is least for RIB. Since the speed of a parallelizer is directly influenced by the total data exchange across processors, it can be seen that the time taken by RIB is also the lowest.
While there is no particular pattern observed in the trend of the RCB method, a distinctive drop in the data exchange can be seen at the middle of the simulation. This may be attributed to the fluid flowing downwards and becoming almost horizontal. The rising trend after that could be attributed to the obstacle, hitting which the flow again becomes turbulent and data-transfer increases.
HSFC has almost invariant data-transfer trend. Also HSFC has the least difference between the max and min particles exchanged while this is highest for RIB. HSFC thus can be said to be uninfluenced by the nature of the problem and domain, while RIB significantly depends on these factors.
RIB is thus proved to be the most efficient method for arbitrary domain problems like dam-break.

7.3 zsph Performance

zsph as the parallelizer is called has been in development since last 6-8 months. Performance analysis of its progress can be seen in this section. Below is the nomenclature used to define particular sub-versions of zsph :

- sph2d : Serial SPH implementation (used as benchmark)
- zsph : First zsph implementation (all properties sent)
- zsph 2 : Only required properties sent
- zsph 3 : Added symmetric interactions

3 seconds dam_break simulation :

Scorpio

Processors \Particles	**5227**	**10424**	**19226**	**29725**
1 (sph2d)	00:29:18	01:33:28	04:15:37	07:40:17
2 (zsph RIB)	20:23.67	54:55.57	02:41:34	04:52:07
3 (zsph RIB)	00:14:45	41:18.17	01:49:35	03:31:37
4 (zsph RIB)	00:12:19	34:13.37	01:24:34	02:47:01
4 (zsph 2 RIB)	–	–	–	01:50:26

Table 6: Scorpio : zsph Performance

Rake & Vorton

Processors \Particles	**5227**	**29725**	**70203**
8 Rake (zsph 2 RIB)	09:15.10	01:37:04	05:57:00
4 Rake + 4 Vorton (zsph 2 RIB)	13:55.91	02:01:29	06:29:29
8 Rake + 8 Vorton (zsph 2 RIB)	–	–	05:01:20

Table 7: Rake & Vorton : zsph Performance

1 second dam_break_obstacle simulation :

Nebula, Rake & Vorton

Processors\Particles	**278855**
8 Rake (zsph 3)	20:00:00 (Approx)
8 Rake + 8 Vorton (zsph 3 RIB)	24:14:00
12 Nebula (zsph 3 RIB)	22:58:38
64 Nebula (zsph 3 RIB)	31:00:00

Table 8: Large problems : zsph Performance

8 Conclusions and Future work

- Current parallel module is approximately 4-5 times faster than the serial implementation
- The module depends on Zoltan for load-balancing and ghost computation
- Geometric partitioners appear to be a very good choice as a load-balancer for parallelizing SPH like CFD methods
- Graph-partitioning implementation would require migration from particle-based partitioning to cell-based partitioning
- Periodic load-balancing to be implemented
- This would require porting away from Zoltan for ghost-computation
- Attempt to implement in-house load-balancer (K-means) to completely eliminate dependence on Zoltan. K-means could be the default load-balancer. Zoltan would be an optional dependency.
- Merge with PySPH to release next version

9 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

[7] http://www.cs.virginia.edu/stream/

[8] https://fs.hlrs.de/projects/par/mpi//b_eff/