Parallel framework for Smooth Particle Hydrodynamics

B.Tech Project

by

Pushkar Godbole (09D01005)

guide

Prof. Prabhu Ramachandran

PySPH

- Open-source Python framework for Smoothed Particle Hydrodynamics (SPH) simulations
- Physically sound and works serially
- Modular structure enables addition of new functions and features
- Parallel module needed for solving large, realistic problems
- Existing parallel module inefficient and slow

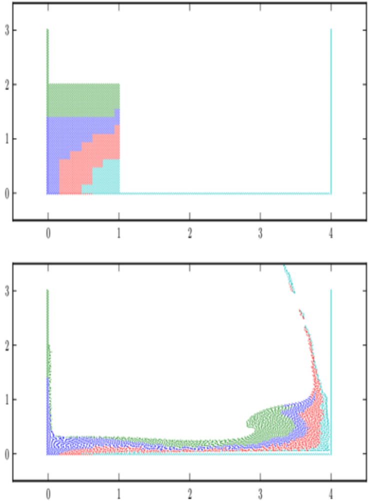
Objective

Develop a parallel module that would,

- be de-linked from the solver. Solver need not “see” the parallel module
- achieve and maintain balanced work-load across processors **(Load-balancing)**
- effectuate load-balancing while minimizing ghost (remote) particles **(Halo-minimization)**
- execute efficient transfer of “export” and “ghost” particles across processors **(Data transfer)**

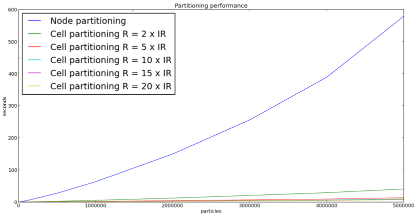
Dynamic Load-balancing

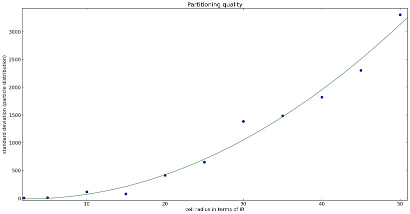
- Steep rise in halo-region size with time
- Dynamic load-balancing inevitable



Cell-partitioning vs Node-partitioning

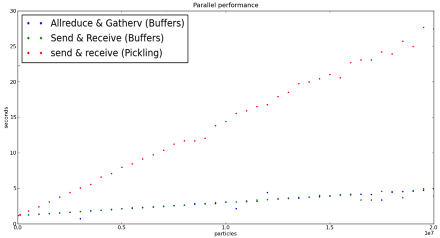
- Highly expensive runtime for node-partitioning
- Cell-partitioning inevitable in the long run
- Serial Metis used for analysis. Need to explore parallel partitioners





Data-transfer

- **mpi4py :** Python wrapper for MPI
- mpi4py's buffer based Send & Receive method most efficient and suitable for our application



Load-balancing

Working implementation has particles as partitioner objects.
Currently load-balancing has to be run at every time-step

- Local and Global ids are assigned to particles
- Load-balancer partitions the problem in parallel across all processors
- Export lists of particles and their destinations generated in every processor
- Data of all export particles transfered to appropriate destinations using the designed data-transfer protocol

Ghost computation

Ghost particles are computed after load-balancing and corresponding data transfer are done

- For each cell, a bounding-box (of thickness 2 x IR) constructed
- Overlap of the bounding-box with processors checked
- Copy of all particles in a cell to be sent to processors overlapping the bounding box
- Ghost (remote) particle exchange list generated and particle data transfered to appropriate destination processors

Data-transfer

Data-transfer occurs at 2 stages : Export particles after load-balancing & Remote particles after ghost-computation

- Requires list of particle gids/lids and corresponding array of destination processors
- All processors execute query to send data to processors with a lower rank
- All processors execute query to receive data from all senders
- All processors execute query to send data to processors with higher rank
- Tried implementing non-blocking queries with unique tags : Didn't work

Zoltan

Zoltan is a C library for parallel partitioning algorithms and data migration tools

- Geometric, Graph, Tree based partitioners available
- Python wrapper for Zoltan (PyZoltan) written by Kunal used in the parallelizer
- PyZoltan used for load-balancing and ghost particle location
- Currently only geometric partitioners used as particles are zoltan objects
- Migrating to cell based objects to facilitate graph-partitioning
- Load-balancing required at every time-step to update the zoltan object map for correct ghost particle location

Periodic load-balancing

Execute load-balancing step periodically only after the distribution gets highly obfuscated. Would require de-linking ghost particle locator from Zoltan. May be implemented as follows :

- Measure total number of ghost particles exchanged at every time-step. If exceeds a threshold, execute load-balancing
- At each iteration, find min and max domain limits of every processor : Construct domain box
- Identify boundary cells (cells with less than 8 neighbors in case of 2D)
- For each boundary cell, construct a bounding box with 2 x IR thickness
- If bounding box overlaps with the domain box of a processor, cell is a remote-cell to that processor

Geometric Partitioners

Zoltan has a multitude of load-balancing techniques. The parallel module currently supports the geometric techniques and would soon support Graph based methods too.

Recursive Co-ordinate Bisection (RCB)

- Begins bisecting the domain using a plane orthogonal to the coordinate axes
- Generates two sub-regions with almost equal number of objects
- Works recursively to reach the desired number of partitions

Recursive Inertial Bisection (RIB)

- Variant of the RCB for non axis aligned domains
- Bisection line is orthogonal to the principle inertial axis

Geometric Partitioners

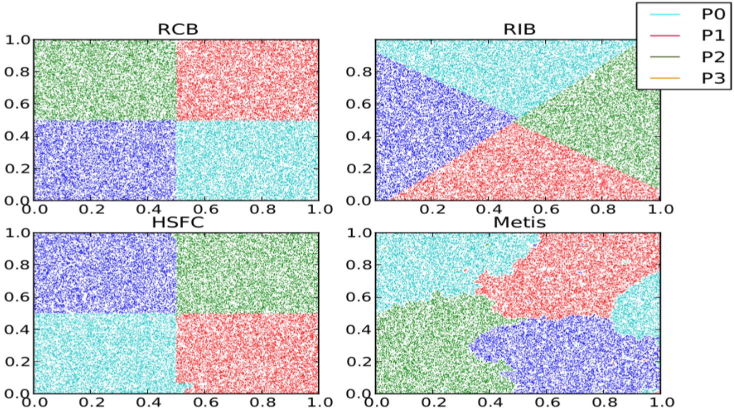
Hilbert Space Filling Curves (HSFC)

- Maps a given point in a domain into 1D interval
- N bins are created to partition the problem space
- Starts from initial point and places cuts when desired weight reached
- Recursively improves with increase in value of N

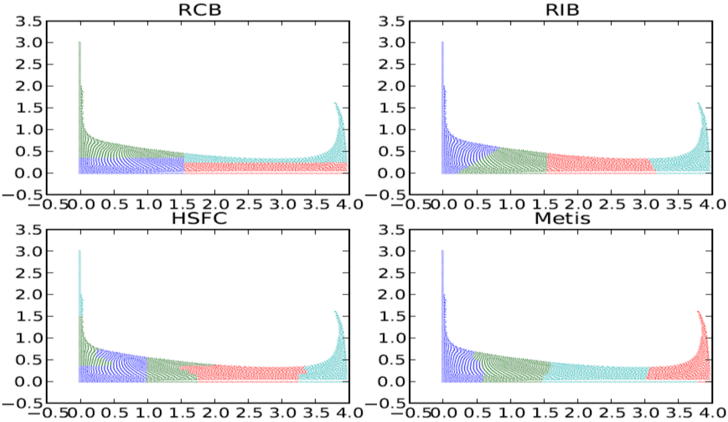
Graph Partitioners

- Mesh/particle data re-interpret as graph with elements as nodes and element connectivity as the edges
- Performs cuts along edges to create sub-graphs with approximately equal weights
- Cell-partitioning inevitable in order to implement graph-partitioning in PySPH
- With cell-partitioning, each object would generally have 8 neighbors in 2D and 26 in 3D

Box particle distribution



Dam-break particle distribution



Symmetric interactions

- In SPH, if A is B's neighbor, B has to be A's neighbor
- By newton's 3rd law, A's influence on B would be equal to B's influence on A
- Use this feature to half number of computations for acceleration computation
- Two techniques used : Gid/Lid based, Cellwise

Gid/Lid based SI

- Neighbor locator returns list of neighbors for every particle
- Out of the list, acceleration contribution taken only from particles with higher Lids/Gids
- Acceleration contribution symmetrically added to both influencing and influenced particles
- Speed-up of 16% observed over older implementation

Cellwise SI

- In Gid/Lid based SI, extraneous neighbor location wasted
- Acceleration computation iterates over cells instead of particles
- Once a cell's acceleration computation done, cell marked with a flag
- Neighbor locator modified to only look into cells with a "False" flag
- Acceleration contribution symmetrically added to both interacting particles
- Speed-up of 10% observed over Gid/Lid based SI

Machines

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

Network characterization

Latency & Bandwidth :

- 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

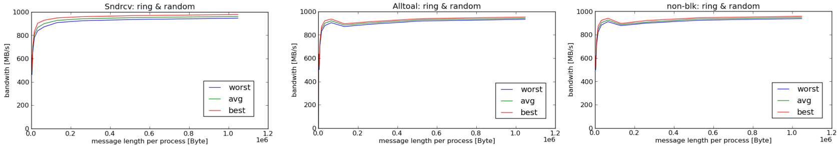
Network characterization

Latency :

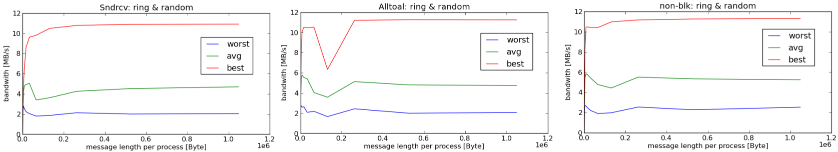
Machine \Parameter	**Latency rings (microsec)**	**Latency Rings & random (microsec)**	**Latency ping-pong (microsec)**
Gemini (4 procs)	1.752	1.748	0.879
Draco (4 procs)	1.522	1.529	0.73
Gemini-Draco (4+4 procs)	36.536	73.176	1.049
Gemini-Draco (2+2 procs)	68.839	103.426	0.596

Network characterization

Bandwidth :



Gemini Bandwidth



Draco-Gemini Bandwidth

Network characterization

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

Network characterization

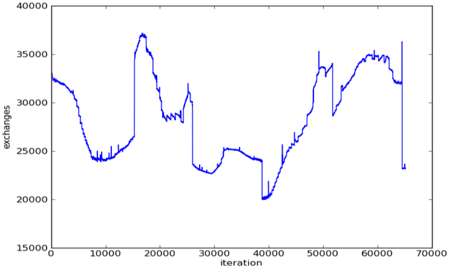
Computation rate :

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

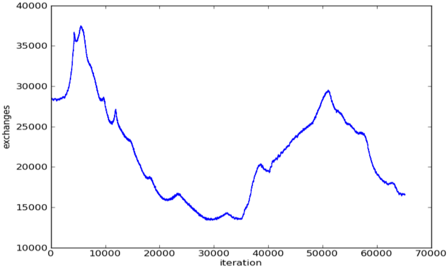
Function	**Best Rate (MB/s)**	**Avg time (sec)**	**Min time (sec)**	**Max time (sec)**
Copy	3479.9	1.386182	1.379369	1.392447
Scale	3488	1.387069	1.376131	1.394406
Add	4533.2	1.594228	1.588294	1.606344
Triad	4528.6	1.593789	1.589878	1.605193

RCB vs RIB vs HSFC

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

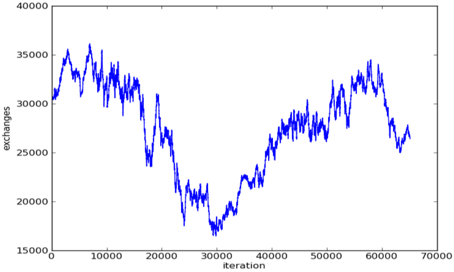


RCB



RIB

RCB vs RIB vs HSFC



HSFC

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

zsph performance

- 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

zsph performance

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

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

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

Conclusions & Future work

- Current parallel module is approximately 4-5 times faster than the serial implementation
- Module depends on Zoltan for load-balancing and ghost-computation
- Attempt graph-partitioning to compare performance with geometric-partitioners
- Attempt to implement in-house load-balancer (K-means) and ghost-particle locator
- Implement periodic load-balancing
- Merge with PySPH to release next version

Thank You!