Implementation of a Load Balancing scheme and Domain Decomposition in the Independent Parallel Particle Layer library

Michael Ligotino, *ETH Zürich*
Advisor: Andreas Adelmann, *Paul Scherrer Institute, PSI / ETH*
Scientific collaborators: Sriramkrishnan Muralikrishnan, *Paul Scherrer Institute, PSI / ETH*
Matthias Frey, *University of St Andrews*

Abstract

We present the implementation of a load-balancing scheme and domain decomposition using an Orthogonal Recursive Bisection [1]. The implementation is verified with several simple tests before showing some benchmark results. The algorithm effectively reduces load imbalance across processes, in order to be beneficial, the performance critical parts are identified for future optimization. This is done as part of the development of version 2.0 using Kokkos [2], which is a C++ library that provides abstractions to allow a single implementation of an application kernel to run efficiently on different kinds of hardware.

1 Introduction

In a simulation using Particle-In-Cell (PIC) in parallel environment with MPI [3], grids and particles need to be divided between processes (MPI ranks). In the library Independent Parallel Particle Layer (IPPL), this decomposition is regardless of particles. This creates or leads to imbalance for non-uniform distributions of particles or during time evolution. This imbalance increases the computation time of certain processes, which affects the overall performance as other processes are idle during that time. In PIC, particles are typically much more than grid points and hence it would be beneficial to load balance with respect to particles. This can be achieved by using an Orthogonal Recursive Bisection (ORB) (fig. 1), where the domain is decomposed by doing a weighted reduction and dividing the grid at the position that halves the total workload. This algorithm will be presented in details in the next sections.

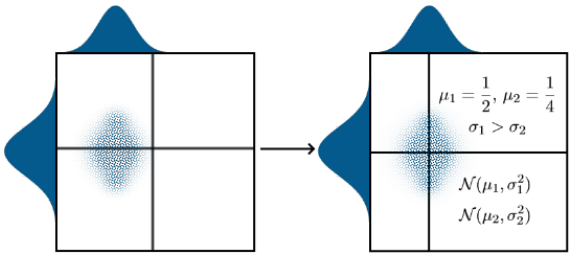

Figure 1: Example of a decomposition in 2D for a normal distribution of particles. The lines are the boundaries between processes.

During the time evolution of the simulation, particles may move among ranks and create again an non-negligible imbalance. The domain decomposition is called whenever the imbalance of any rank excesses a certain threshold. The choosing of this threshold can improve performance as will be showed in later sections.

2 Domain Decomposition

2.1 Preliminary remarks

IPPL is a framework that provides data structures and operations for particles and fields.
A grid size, the number of particles, and the number of time steps is given to IPPL. From this, it creates a mesh, a layout, and particles (fig. 3). Within this project, a charge density and position distribution are assigned to the particles. Such data are stored in Fields, which are multi-dimensional arrays (Kokkos views), divided between processors.
Particles are also Kokkos views, local to the processor that holds the relevant portion of the Field. Figure (2) shows how IPPL divides subdomains among ranks.

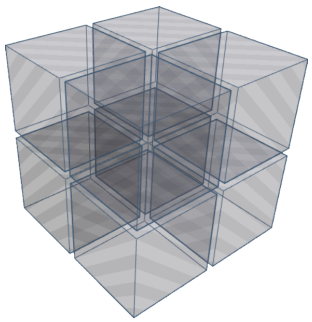

Figure 2: Initial cutting of cubed domain, example for 8 ranks.

The processor can be on host or on device, respect to whether the processing unit is a CPU or a GPU respectively. Communication is needed to send these particles to the appropriate ranks.
An important property in IPPL are the halo cells, which handle ghost cells. Those are layers which can be found between ranks' boundaries, and at the boundaries of the whole domain. To get the relevant data, these layers must not be used in physical process.

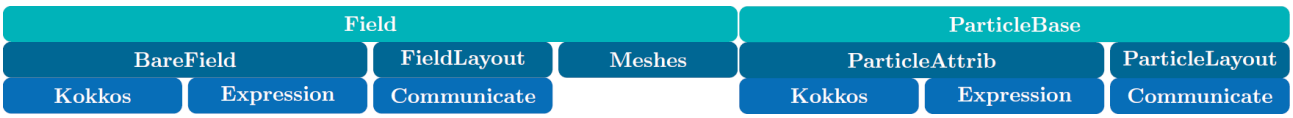

Figure 3: Structure of particles in IPPL 2.0
Source: Matthias Frey

2.2 Orthogonal Recursive Bisection

A domain is described by three set of indices $[i : j]$, $i, j \in \mathbb{N}, 0 \leq i < j$, and $j <$ grid size. In algorithm (1), the important steps of the decomposition are shown.
First, the position of particles are scattered on a member field using Cloud-In-Cell interpolation. The repartition of the domain can thereafter start using an ORB. The particle density is reduced orthogonally along the longest axis of the domain, the reduced value on each plane is put in an array (fig. 4). When all the planes have been reduced and the array is filled, the location of cutting is determined by the median of that array. The set of indices along the cut axis is split at that position. This is done recursively until a single process is assigned to a subdomain.

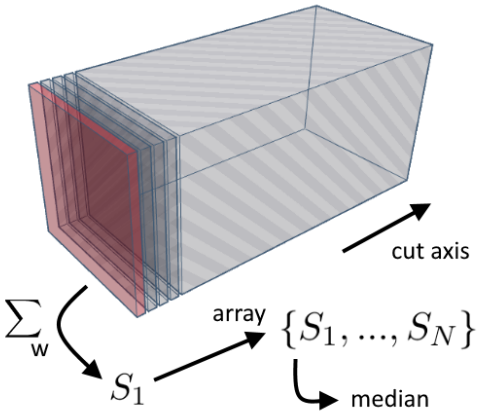

Figure 4: Perpendicular reduction of a rectangular shape. The sum of weights S_i, $i = 1, ..., N$, is stored in a array of size given by the length of the cut axis.

The ORB works with $n = 2^m$ MPI ranks, where m is an integer. Since each time the layout is cut in two, there are $n - 1$ cuts in total.
The total data is split among all ranks, therefore, after the local reduction, each rank must know where to cut the set of indices to split the domain. This communication, although expensive, is necessary. This step is shown in algorithm (2) which presents the perpendicular reduction of the domain. Since subdomains may have any set of indices, the reduction of a subdomain must be done only within the set of indices that are included in the rank (fig. 5).

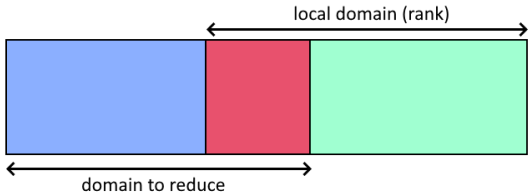

Figure 5: Intersection (red) between domain to reduce and local domain of a rank showing the relevant part to reduce.

Algorithm 1: BinaryRepartition

Data: ParticleAttrib r, FieldLayout fl
Result: Performs an ORB on a layout given a set of weights

```
   /* Scatter particles' positions in
      member field                          */
[1] bf_m ← scatter(r);
   /* Set variables                         */
[2] nprocs ← total number of ranks;
[3] doms[] ← array of subdomains;
[4] procs[] ← no. of procs per subdomain;
   /* Start with whole domain and total
      number of procs                       */
[5] it ← 0;
[6] doms[it] ← fl.getDomain();
[7] procs[it] ← nprocs;
   /* Cut recursively each domain until a
      single processor is assigned to each
      subdomain                             */
[8] mproc ← nprocs;
[9] while mproc > 1 do
[10]    cutAxis ← longest axis of doms[it];
[11]    reduced[] ← PerpReduction(cutAxis,
         doms[it]);
[12]    cutLoc ← median(reduced);
        /* Split domain at cutLoc along the
           cutAxis, store them in left and
           right subdomains                 */
[13]    ldom, rdom ← doms[it].split(cutAxis,
         cutLoc);
        /* Update new subdomains and divide
           current processors count         */
[14]    doms[it] ← ldom;
[15]    doms.insert(it+1, rdom);
[16]    procs[it] ← procs[it] ÷ 2;
[17]    procs.insert(it + 1, procs[it]);
        /* Find biggest processor count and
           reassign the subdomain to cut as
           that one                         */
[18]    mprocs ← max_value(procs);
[19]    it ← max_index(procs);
[20] end
   /* Update FieldLayout with new domains
      */
[21] fl.updateLayout(doms);
   /* Update member field with new layout
      */
[22] bf_m.updateLayout(fl);
```

Algorithm 2: PerpReduction

Data: int cutAxis, NDIndex dom
Result: Performs an orthogonal reduction on a domain along the cut axis

```
[1]  ldom ← bf_m.getLocalDomain();
[2]  data ← bf_m.getLocalData();
[3]  reduction[] ← array of reduced weights;
     /* Determine the bounds on which to sum
        the weights for the three axes
        (cutAxis, two perpendicular axes)   */
     /* For d=cutAxis, d=perp1, ...         */
[4]  inf_d ← max(ldom[d].first(), dom[d].first());
[5]  sup_d ← min(ldom[d].last(), dom[d].last());
[6]  for i ← inf_cutAxis to sup_cutAxis do
[7]      weights ← 0;
[8]      for j ← inf_perp1 to sup_perp1 do
[9]          for k ← inf_perp2 to sup_perp2 do
[10]             weights + = data(i,j,k);
[11]         end
[12]     end
[13]     reduction.append(weights);
[14] end
[15] treduction[] ← array of total reduction;
     /* Communicate with others the value of
        each of the rank and sum them       */
[16] MPI_Allreduce(treduction, reduction,
      MPI_SUM, ...);
[17] return treduction;
```

Once the domain decomposition is finished, all fields must be updated with the new layout, and the particles must be redistributed, i.e. sent to the corresponding rank.
The algorithm does not have any condition on the final size of the subdomain, it can cut a domain in a subdomain of length 1, i.e. a plane. However, it appears that sending particles to a rank with grid size 1 along any of the dimensions leads to problems. That is why ORB for IPPL is adapted so that no planes are produced in the decomposition, this is done in the choosing of the median by simply shifting the median by 1 index when it is near a boundary.

2.3 Threshold

During the time evolution of the application, the imbalance on each rank is measured. The latter consists in checking the deviation from the ideal share of particles per rank, which is simply the total number of particles divided by the number of MPI ranks. If this deviation is larger than a certain threshold, then the domain must be decomposed for all ranks.

3 Verifying the decomposition

Simple tests are done to verify the correctness of the decomposition.
First, to confirm the correctness of the reduction, a particle is positioned on each grid point. Other tests using uniform and normal distributions are provided to check whether the final decomposition is as expected, this is done mostly visually for high number of ranks.
All tests on CPU from 2 to 64 ranks, and on GPU from 2 to 8 ranks are done on an NVidia DGX A100 (40 GB) [4]. Tests with higher number of ranks are done on Intel Xeon Gold 6152 processors (2.10 - 3.70 GHz) [5].

3.1 Particles on grid points (PGP)

In the case of PGP, the number of total particles must match the number of grid points. For each test in this section the distribution of particles along the cut axis (example: top plot in fig. (6)) is shown, moreover the total amount of particles on each side of the cut (example: bottom plot in fig. (6)) is shown. In the case of PGP, it is expected to have exactly half of the total particles on each side.

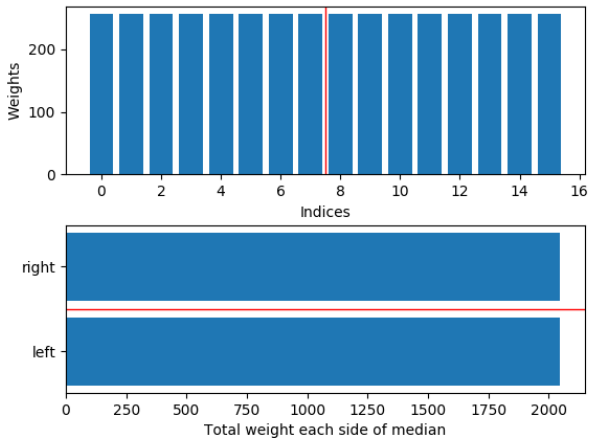

Figure 6: Histogram of weights for PGP with 4 ranks and size 16 cubed, i.e. 4096 particles, each rank possesses 1024 particles distributed among 8 indices. Therefore on each index there are 128×2 particles. (see fig. 7)

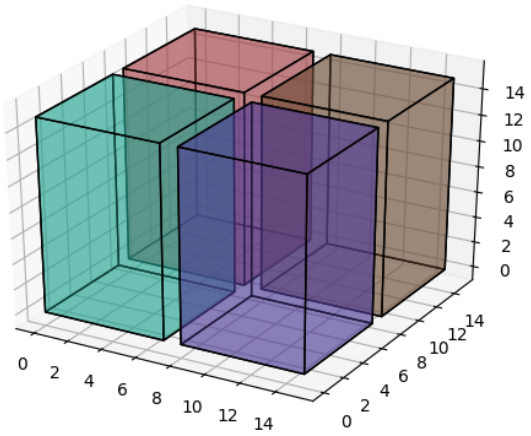

Figure 7: Visualization of the final decomposition for PGP with 4 ranks and size 16 cubed.

These tests (fig. 6) confirm that the reduction works properly. Indeed, by getting the right amount of particles it is clear that no ghost cells are summed over, and moreover that the cutting is correct.
In figure (8), as can be seen in figure (9), there are twice as many ranks along one direction, than in the other two. Therefore tests like PGP help confirm that the set of in-

dices on which the reduction happens is the intersection between the local domain and the domain to reduce.

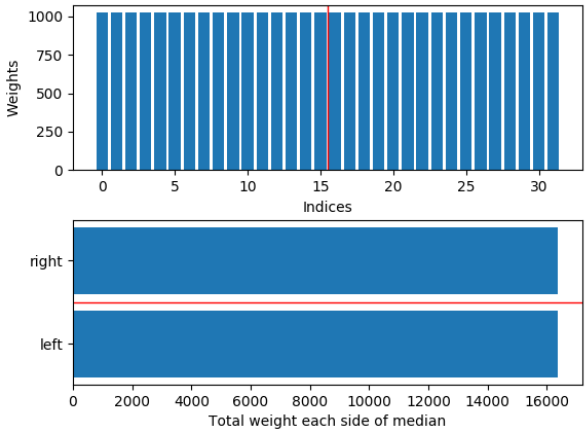

Figure 8: Histogram of weights for PGP with 16 ranks and size 32 cubed, i.e. 32768 particles, each rank possesses 2048 particles distributed among 8 indices. Therefore on each index there are 256×4 particles. (see fig. 9)

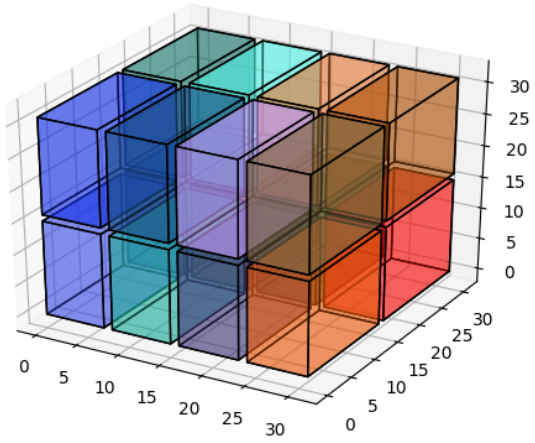

Figure 9: Visualization of the final decomposition for PGP with 16 ranks and size 32 cubed.

3.2 Uniform/Normal distributions

Histograms of weights for a single cut is irrelevant here as there are many ranks. The idea is to check whether the algorithm gives a final decomposition which is as expected, given a particle distribution.
Only two cases are shown here: the case of a gaussian distribution, and that for a uniform distribution on restricted range.
As can be seen in figure (10) there is a denser number of subdomains around the mean μ. The same can be said for figure (11) where particles are uniformly distributed between 0.2 and 0.6 in the three axes.

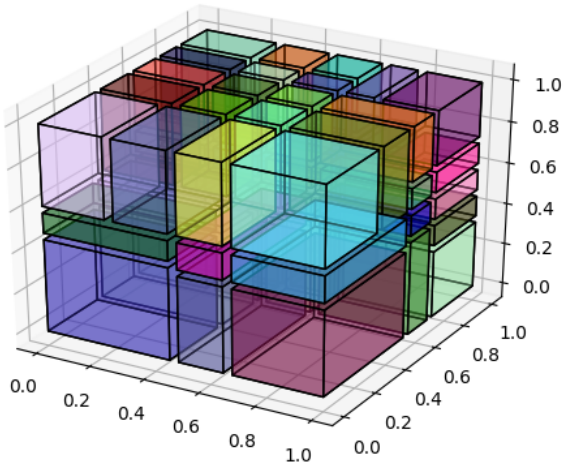

Figure 10: Visualization of the final decomposition for $\mathcal{N}(\mu = (0.5, 0.75, 0.6), \sigma = (0.3, 0.2, 0.2))$ with 64 ranks and size 32 cubed.

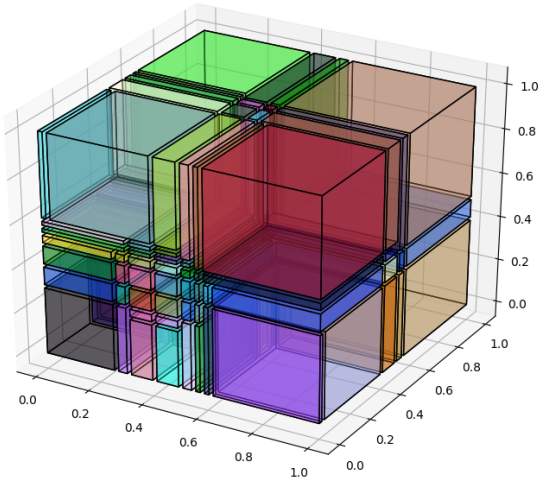

Figure 11: Visualization of the final decomposition for $\mathcal{U}(0.2, 0.6)$ with 256 ranks and size 64 cubed.

3.3 Conserved quantities

When repartitioning the layout throughout the time evolution of the simulation, it is important to check that no particles are "lost" and moreover, that the layout keeps the same volume. The number of particles is checked externally of ORB, simply by summing the local number of particles in each rank via an MPI collective. The volume conservation can be checked by summing the volume of each ranks' local domain. Each rank has a volume normalized with the volume of the global (non-local) domain, therefore at each time step, the sum of each of the volumes must be 1.0. In figure (12) the conservation of the total volume is shown, this is tested by moving particles in a random direction but by no more than one grid cell. This has been tested as well on CPU up to 128 ranks.
The charge conservation is also tested after the domain decomposition as a sanity check. This is done externally of ORB as well.

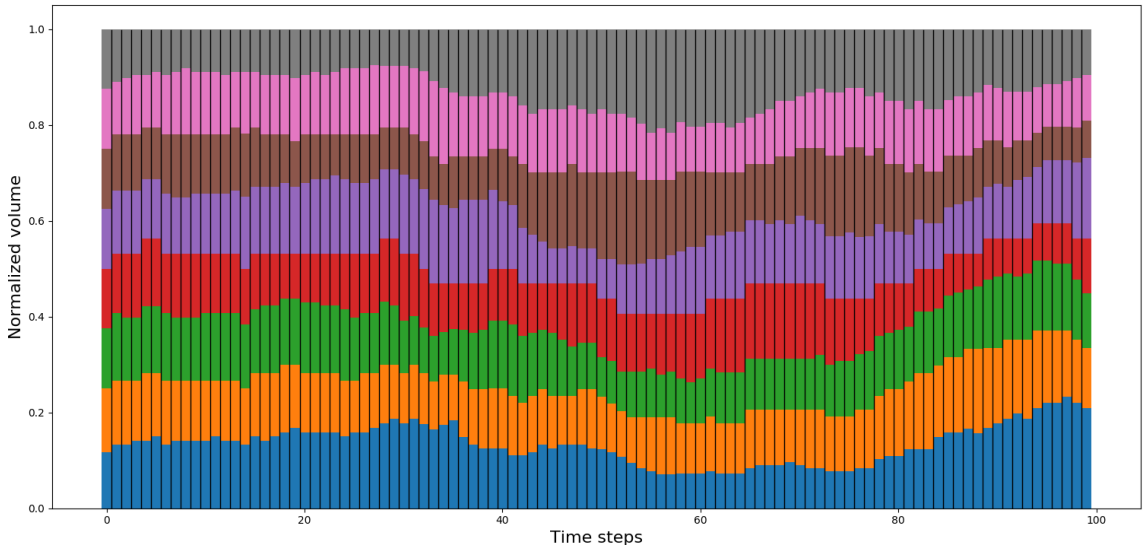

Figure 12: Conservation of volume for 8 ranks (GPU), a 32 sized cube and for 100 time steps. Each color represents a rank.

4 Performance

In this section, the performance critical parts are identified by analysing timings and load imbalance of various problem configurations.

4.1 Load balancing

In this section, several results are presented with a normal distribution $\mathcal{N}(\mu = (0.5, 0.75, 0.6), \sigma = (0.3, 0.2, 0.2))$. The deviation from the ideal share of particles is measured for each rank, the total is added in a histogram. Therefore, the less this deviation, the better the load is balanced. When running the simulation without repartitioning the layout, figure (13) shows the result. As the particles move randomly and by at most one grid-cell, also because of the boundary conditions, particles are expected to "diffuse" in the whole domain, which explains the decrease of load imbalance.

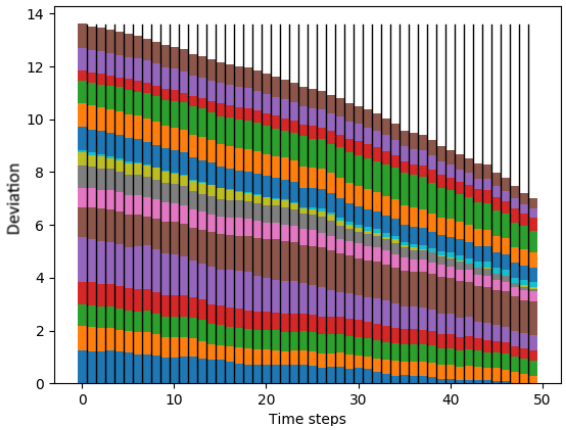

Figure 13: Imbalance per rank, PIC with 50 time steps, for 16 ranks and 128 cubed grid.

The same simulation is done with the load balancing enabled (fig. 14). As can be seen in the measure of deviation, the load imbalance has effectively reduced by 10. Since these simulations were done with a threshold of 15%, it is interesting to check whether changing this threshold improves the total average time of the simulation.

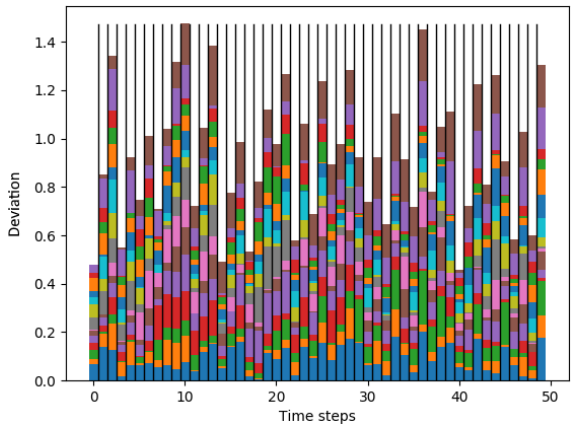

Figure 14: Imbalance per rank, PIC with 50 time steps, 16 ranks and 128 cubed grid.

Figure (15) shows the timings for thresholds from 10% to 25%. Clearly, a value too low is going to lead to many decompositions, which is too expensive. And a value too large will leave more imbalance, leading to an increase of the simulation time. As can be seen in the picture, a value of 18% seems to be good.

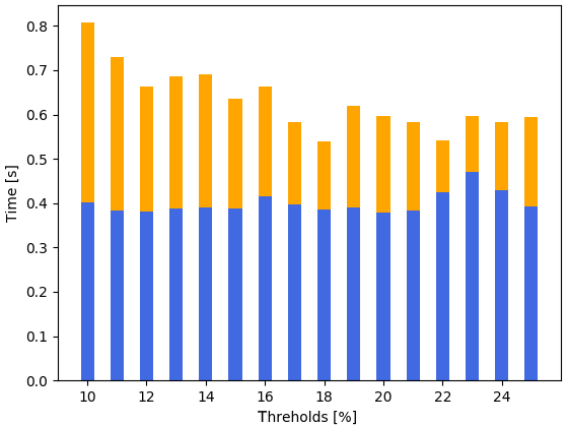

Figure 15: Average total timings with the amount used for repartitioning (orange), PIC with 50 time steps, 16 ranks and 128 cubed grid.

4.2 Timings

In this section, timings for different configurations are shown in the case of a uniform distribution of particles. In figure (16) the increase of total running time when increasing the size of the grid for fixed ranks can be seen. This is the timing for a single (initial) domain decomposition.
Notice that GPUs seem to handle much better the increase of the grid size. Also at large grid size, the strange decrease of overall time for increasing ranks in CPU versus the increase of time for increasing ranks in GPU.

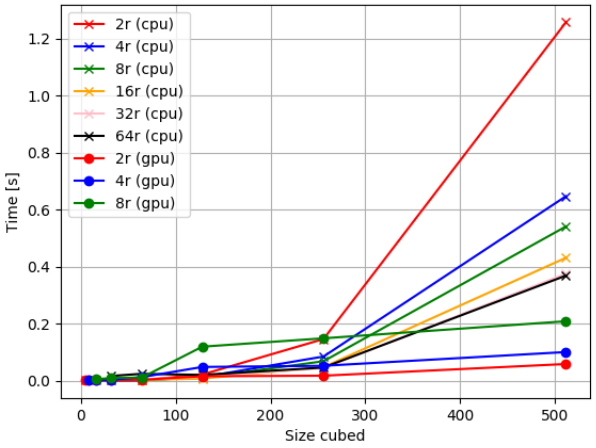

Figure 16: Timings comparison for ranks from 2 to 64 on CPU, and 2 to 8 on GPU.

The cost of repartitioning can be measured by comparing to the overall timing of the simulation. In figure (17) can be seen the normalized timings, with the time percentage used for decomposing the domain throughout the simulation using a threshold of 15%. With such a typical threshold, it is not expected to repartition at each time step.

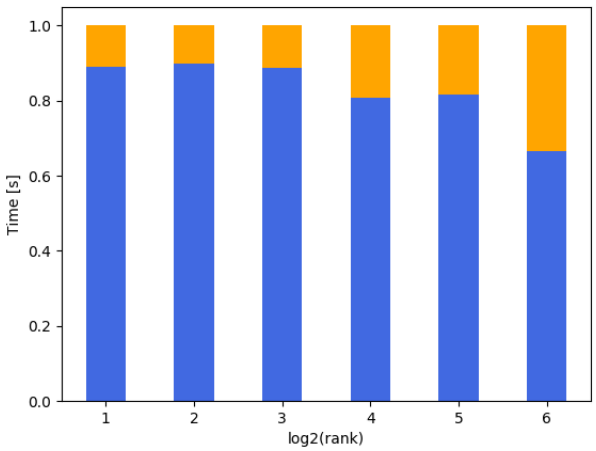

Figure 17: Comparison of the portion used for repartitioning (orange) for different ranks (from 2 to 64) on CPU in a PIC simulation with increasing sizes (from 8 to 256 cubed) and 10 time steps.

Clearly, domain decomposition requires more time as the size and number of ranks increase. It becomes apparent in this case that the cost of repartitioning must be optimized so that it does not increase exponentially. In figure (18) can be seen the portions of workload in a domain decomposition step, for the case of few ranks and small grid size. The most expensive part of the algorithm are the scattering, the perpendicular reduction, and the communication, where the latter also accounts for redistributing the particles among ranks. Moreover the pie chart shows the portion used for updating the layout with the new indices. All the other negligible parts of the algorithm are in "basic operations".
As can be seen, the scattering of particle density on the member field is taking a non-negligible part of the work. Almost all of the rest is in communication. Indeed, the update of the particles (PB), which is a required step after the decomposition (outside of ORB), is taking most of the work in this case. The communication between ranks (all reduce) of the location of the cuts, is as expected increasing with the number of ranks (fig. 19). The latter represents $\approx 80\%$ of the total timing, which is too expensive for a simulation.

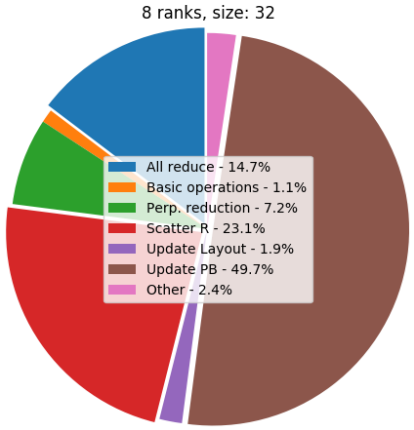

Figure 18: Portion of work for a single domain decomposition step in the case of 8 ranks on CPU with size 32 cubed.

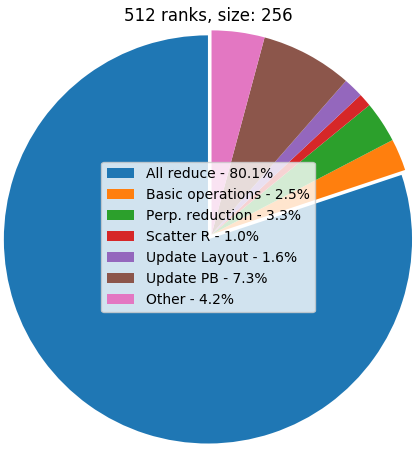

Figure 19: Portion of work for a single domain decomposition step in the case of 512 ranks on CPU with grid size 256 cubed.

Back to figure (16), it is interesting to check whether the behaviour of these portions of work change in the extreme cases (512 cubed size), the case of 2 ranks for CPU is compared to that of 2 ranks for GPU below.

Figure (20) shows the strange behaviour of the portions of workload in the process of reparitioning. With 2 ranks, cleary the communication part is small. However the update of layout takes most of the workload, this represents (by checking with figure (16)) $\approx 0.85\,\mathrm{s}$. By comparing with the behaviour of GPUs in the same configuration (fig. 21), it can be seen that updating the layout is also increasing but better handled, as this represents only $\approx 0.016\,\mathrm{s}$ for GPUs.

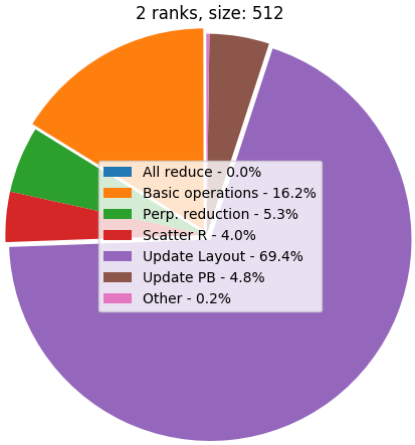

Figure 20: Portion of work for a single domain decomposition step in the case of 2 ranks on CPU with size 512 cubed.

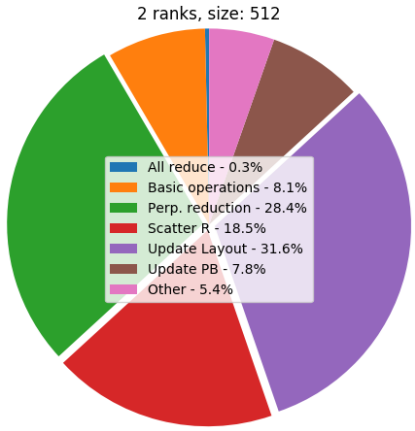

Figure 21: Portion of work for a single domain decomposition step in the case of 2 ranks on GPU with size 512 cubed.

5 Discussion and improvements

The advantage of the Orthogonal Recursive Bisection is its simplicity. It is clear how it operates. However, its implementation in IPPL comes with a few caveats.
From the point of view of performance, the implementation of ORB in IPPL presents several weaknesses.
The first simple source of optimization is in the scattering of weights. This is clear in timings for small ranks and small grid sizes, where this single operation is more expensive than the rest of the decomposition. The particle-to-mesh method used here is Cloud-In-Cell, an optimization would be to use Nearest-Grid-Point.

The reduction portion grows with each axis a of the layout, meaning an increase of the axes has a complexity of $\mathcal{O}(|a_1|\cdot|a_2|\cdot|a_3|)$. This cost can be clearly seen in the increase of timings both in terms of computation timing, and in communication (all-to-all). Since an array of the size of the cut axis is sent and received for each rank, the amount of communicated data is of the order $\mathcal{O}(|a_{cutAxis}|(n-1)n)$ where n is the number of ranks. However, there are many possibilities ahead to improve the MPI collectives. An attempt to reduce the timings was to change the amount of communications to an order $\mathcal{O}(|a_{cutAxis}|2(n-1))$ by selecting a single rank as the root, which would compute the median of the array, to finally broadcast it. However, in this case, the timings are roughly the same. In this direction, a possible improvement would be to group involved ranks in the reduction in a specific MPI communicator, therefore the all-to-all would only refer to those within that group. However, processes that are not involved must still know about the location to split the domain, meaning communication from the group to the outside is necessary. Another more convincing possibility is to decrease the amount of sent data, indeed, as the useful data is smaller with each step of the ORB, many null values are sent for nothing. In this direction, it could be interesting to use a predefined MPI_Datatype for this purpose, which could enhance the MPI collective performance.
Now letting aside the algorithm per se, an expensive task which happens after the domain decomposition is the update of particles, i.e. send them to the corresponding processing unit. In an ideal HPC simulation, particles should be sent to the nearest neighbouring processors physically, ideally avoiding inter-node communication. Although this is not implemented in IPPL, it could lead to an improvement in the cost of updating the particles. Indeed, as the simulation goes on, particles are moved and updated, when repartitioning, particles are updated again. That is, whenever ORB is called, particles are updated two times in the same time step. This can explain the long timings of the simulation.

6 Conclusion

The initial task of implementing an ORB for domain decomposition and load balancing in version 2.0 of IPPL was achieved. It greatly reduces load imbalance. Although a couple of modifications to IPPL were necessary to implement it successfully, the algorithm was written as independent as possible.

References

[1] Florian Fleissner and Peter Eberhard. Parallel load-balanced simulation for short-range interaction particle methods with hierarchical particle grouping based on orthogonal recursive bisection. International Journal for Numerical Methods in Engineering, 74(4):531–553, 2008.

[2] Edwards, H. & Trott, Christian. (2013). Kokkos: Enabling Performance Portability Across Manycore Architectures. 2013 Extreme Scaling Workshop (xsw 2013). 18-24. 10.1109/xsw.2013.7.

[3] Walker DW (August 1992). Standards for message-passing in a distributed memory environment. Oak Ridge National Lab., TN (United States), Center for Research on Parallel Computing (CRPC). p. 25. OSTI 10170156. ORNL/TM-12147. Retrieved 2019-08-18

[4] Specifications NVidia-DGX-A10, https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-dgx-a100-datasheet.pdf

[5] Specifications Intel Xeon Gold 6152, https://ark.intel.com/content/www/us/en/ark/products/120491/intel-xeon-gold-6152-processor-30-25m-cache-2-10-ghz.html

A Appendix

A.1 Build

In order to run ORB, first IPPL must be installed (https://gitlab.psi.ch/OPAL/Libraries/ippl/-/wikis/Install). Respect to the installation the user chose, several builds are available: OpenMP, CUDA, Serial.

ORB is already included in any IPPL test file. Create a test file Test.cpp in the test folder. You must include Ippl.h. The simplest way to build the test file is to open the CMakeList.txt file in the test folder, and add

```
add_executable (Test Test.cpp)
target_link_libraries (Test ${IPPL_LIBS})
```

now go in any of the build folder, and build the test file with the following command

```
make Test
```

If the installation of IPPL was correct, the test file should be built.

To add ORB in the test file, it is more comfortable to add the following lines at the top of the file

```
typedef ippl::UniformCartesian<double, 3> Mesh_t;
typedef ippl::OrthogonalRecursiveBisection<double, 3, Mesh_t> ORB;
```

where the 3 refers to the dimension. Now an ORB object can be created anywhere, from which methods can be called.

```
ORB orb;
```

Before being able to call the domain decomposition in ORB, the class must be initialized the FieldLayout and Mesh, as an example:

```
void initializeORB(FieldLayout_t& fl, Mesh_t& mesh) {
     orb.initialize(fl, mesh);
 }
```

Now methods from ORB can be called.

A.2 Run

To run a test file, the easiest way is to use a Workload manager as Slurm (https://slurm.schedmd.com/documentation.html). With the installation of IPPL, sample jobscripts are provided. Should you need to change the number of MPI ranks, the following values must be changed:

```
#SBATCH --ntasks=2
#SBATCH --ntasks-per-node=2
```

When running the test file PIC3d.cpp, the command in the jobscript reads

```
srun ./PIC3d 128 128 128 100000 100 --info 10
```

the first 3 values are the grid size in each axis, the fourth value refers to the number of particles, the fifth to the time steps. The info flag can be removed or edited to display less information by decreasing from 10 to 0.

A.3 Postprocessing

To plot conservation of volumes, domains, histograms of weights, or timings, use the script files given in https://gitlab.psi.ch/AMAS-members/ippl-ligotino. These file should be copied to the build folder used. In each of the script file is described how to obtain the appropriate values for the correct plotting, C++ code is provided at the end to be put in the appropriate part of the class ORB.

As an example, create a script file `script.py` in any build. First load the python module. For instance:

```
module load Python/3.7.4
```

The script can then be run with the following simple command

```
python3 script.py
```