Massively Parallel MPI Implementation of the SPH Code DualSPHysics

Athanasios Mokos[a], Benedict D. Rogers[a]

School of Mechanical, Aerospace and Civil Engineering,

University of Manchester, Manchester, M13 9PL, UK

Abstract

This report covers the implementation of Message Passing Interface (MPI) and the Zoltan library in the Smoothed Particle Hydrodynamics (SPH) code DualSPHysics in preparation for massive parallelisation. The implementation was performed in two stages. During the first stage a new buffer system was developed to allow for asynchronous communication between nodes and data handling was altered to allow for a minimised memory footprint (Workpackage 1). The asynchronous communications were used to overlap node communication with the particle computation minimising idle time. A geometric domain decomposition scheme was created based on domain slices along the main axis. The scheme was used to create a data exchange system between nodes which was used both for creating the neighbour list and for identifying particles moving between nodes (Workpackage 2). The second stage of the project was the integration of the Zoltan library including its communication module and the Hilbert space-filling curve to create a cell map while maintaining spatial locality (Workpackage 3).

Introduction

Smoothed Particle Hydrodynamics (SPH) ([1], [2]) is revolutionising the field of hydrodynamics simulations where its meshless nature implicitly captures the nonlinear deformation of violent motions, such as wave breaking, and obviates the requirement for expensive meshing. SPH is rapidly approaching maturity and is continually developing offering the stability, adaptability and accuracy required in real engineering applications.

The existing DualSPHysics code [3] is one of the most widely used open-source SPH software for these applications with thousands of downloads. DualSPHysics can be run on either a multi-core central processing unit (CPU) or a graphics processing unit (GPU) and is therefore a CPU-GPU code: the flow is simulated either with a CUDA-enabled GPU or with an OpenMP C++ code. Most of the current research, however, has focused on developing the GPU functionality. Taking advantage of a massively parallel architecture with the C++ code, although optimised for use in a single node currently is unsuitable for use beyond small applications. This limits the uptake and applicability of the software as many users in industrial companies who have not invested in GPUs have access to traditional high-performance computing (HPC) clusters.

SPH has already been proven suitable for large-scale MPI applications. Oger et al. [4] and Guo et al. [5] have created MPI SPH codes applicable to thousands of cores. Their main difference is the domain decomposition method: Oger et al. [4] use the Orthogonal Recursive Bisection method [6] while Guo et al. [5] use the Hilbert Space Filling Curve (HSFC) [7]. For this project, the HSFC will be used for domain decomposition; its selection was based on the relative simplicity of its algorithm and the spatial locality maintained when storing memory values. The latter is particularly important for SPH computations as a neighbour searching algorithm is used.

Work Phases

The original project was planned in three phases:

Phase 1: Implementation of MPI functionality to existing DualSPHysics code

Phase 2: Implementation of non–structured (asynchronous) communications.

Phase 3: Improvement of code scaling for thousands of nodes

Phase 4: Application and Dissemination

In this report, a brief introduction to SPH and DualSPHysics will be given. The implementation of MPI and different domain decomposition systems (geometric and HSC) will be detailed. The report will then focus on the algorithms for overlapping communications, particle and halo exchange. Computational and runtime results will be presented.

SPH and DualSPHysics

As a Lagrangian method, Smoothed Particle Hydrodynamics (SPH) is a Lagrangian method that simulates a domain as a set of particles. Function values for each particle are calculated through an interpolation with a weighting function, referred to as a smoothing kernel *W*. The weighting depends on the pairwise particle distance and is controlled by a distance referred to as the smoothing length *h*.

$$\langle A(\boldsymbol{r})\rangle = \int_{\Omega} W(\boldsymbol{r}-\boldsymbol{r}',h)A(\boldsymbol{r}')\,\mathrm{d}\Omega \qquad (1)$$

where A is a function, $\boldsymbol{r}$ is a position vector, Ω is the interpolation domain, and $\langle\ \rangle$ represents the SPH approximation. In a discrete SPH form, this is approximated as ([8],[9]):

$$\langle A(\boldsymbol{r})\rangle = \sum_{j=1}^{N} W(\boldsymbol{r}-\boldsymbol{r}_j,h)A_j\frac{m_j}{\rho_j} \qquad (2)$$

where *N* is the number of neighbouring particles within the kernel radius *2h* and m_j/ρ_j is the volume of the neighbouring jth particle with *m* being the mass and *ρ* the density of the particle.

SPH is used here to simulate flows described by the Navier–Stokes equations for continuity and momentum. In Lagrangian form, these equations are:

Continuity:
$$\frac{\mathrm{d}\rho}{\mathrm{d}t} = -\rho\nabla.\boldsymbol{u} \tag{3}$$

Momentum:
$$\frac{\mathrm{d}\boldsymbol{u}}{\mathrm{d}t} = -\frac{1}{\rho}\nabla p + \boldsymbol{g} + \frac{1}{\rho}\left(\nabla \cdot \rho \nu_o \nabla\right)\boldsymbol{u} \tag{4}$$

where *ρ* is density, ν_0 is laminar viscosity, ***u*** is velocity, *p* is pressure, ***g*** is the gravity vector and *t* is time. Variationally consistent forms of the velocity divergence and pressure gradient are used to derive the SPH approximation of the Navier-Stokes equations [10]:

Continuity:
$$\left\langle \frac{\mathrm{d}\rho_i}{\mathrm{d}t} \right\rangle = \rho_i \sum_j \left[\left(\boldsymbol{u}_i - \boldsymbol{u}_j\right) \cdot \nabla_i W_{ij} \frac{m_j}{\rho_j} \right] \tag{5}$$

Momentum:
$$\left\langle \frac{\mathrm{d}\boldsymbol{u}_i}{\mathrm{d}t} \right\rangle = -\sum_j m_j \left(\frac{p_i + p_j}{\rho_i \rho_j} + \Pi_{ij} \right) \nabla_i W_{ij} \tag{6}$$

An additional equation is needed to link the density and pressure and close the equations.

In addition to the main equations described above, the code used in this report, DualSPHysics, includes several additional models, including viscosity models, a density diffusion model, a symplectic and a Verlet time stepping scheme while boundaries are modelled using the dynamic boundary model [3].

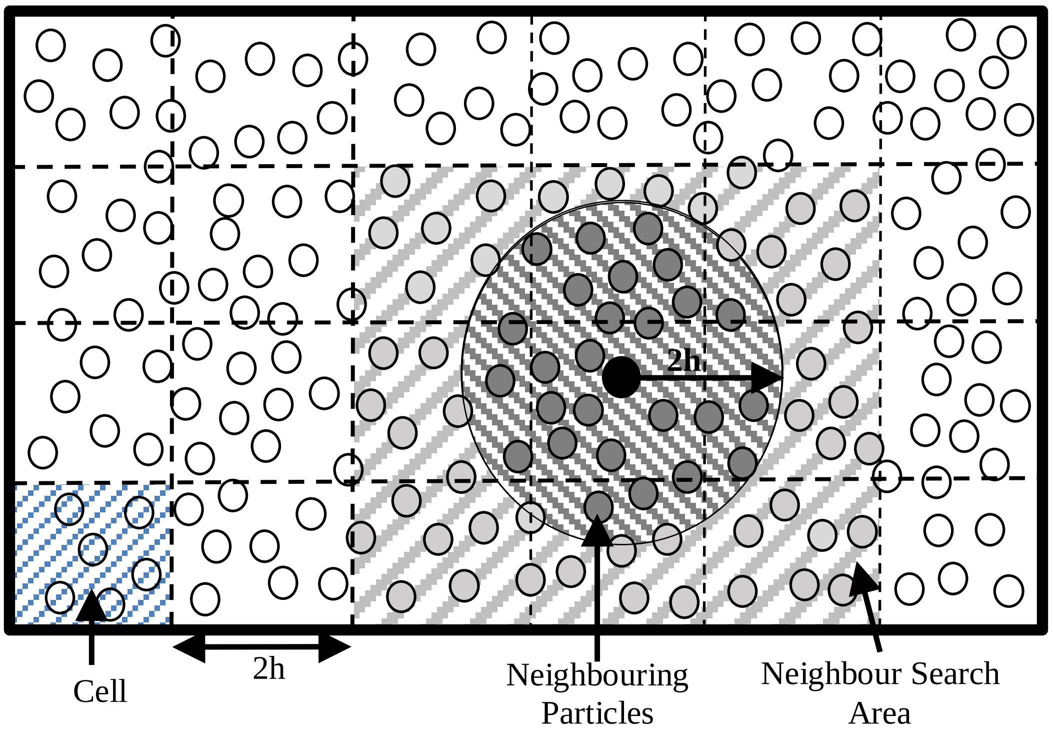


Figure 1: Cell linked list

DualSPHysics is an open source hardware accelerated Smoothed Particle Hydrodynamics code developed for solving free surface flow problems and released under the terms of GNU Lesser General Public License (LGPL). It consists of a set of C++, CUDA and Java codes. A more detailed description of the code can be found in [3].

Of particular relevance for this project for this report is the internal structure of the code. The code can be split in 3 main steps: the Neighbour List (NL), the Force Computation (FC) and the System Update (SU). The NL step uses a Cell-Linked-List (CLL) method described in more detail in [11]. In this method, the domain is divided in square cells of side 2*h* as shown in Figure 1 where each cell has a global ID number. A particle list according to the cell they belong is then created which is used to reorder the variables. A cell list is not created as neighbouring cells can be immediately computed as the domain division is geometric. The algorithm can be seen in Figure 1.

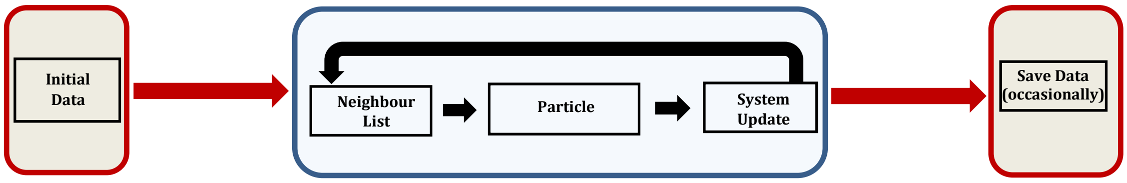


Figure 2: Flow diagram showing the base CPU implementation in DualSPHysics where the repeating key steps taking place on the GPU: generation of the Neighbour List (NL), Force Computation (FC) for the Particle Interaction, and System Update (SU)

The particle list is then used during the FC step to identify particles of the same and adjacent cells. The particle interaction then occurs between these particles rather than the entire domain. Finally, during the SU step variable arrays are updated with the new values. A schematic of the implementation can be seen in Figure 2.

MPI implementation

DualSPHysics can be compiled either for CPU or GPU. The GPU implementation is based on the CUDA/C++ extension while for the CPU the C++ language with OpenMP directives is used. There is however, a large difference in the capabilities of the two implementations stemming from the degree of parallelisation available in each. For the GPU, over 500 CUDA cores are commonly available; in comparison, 8-16 CPU cores per node are usually available through OpenMP. While CUDA cores are have less computational power individually, the greater numbers and degree of parallelism create the gap in performance [12].

The introduction of MPI in DualSPHysics, the main objective of Workpackage 1 (WP1), is expected to bridge this gap in performance. A hybrid OpenMP-MPI implementation will be preferred, with the MPI handling communications between nodes and OpenMP handling communications between cores within a node. As all SPH data are on shared memory, there is no need for explicit data exchange between OpenMP processes.

The integration of MPI necessitated the creation of a new buffer system to enable internode communications and data exchanges. An example of data exchange is shown in Figure 3. Several files were altered and new files were created; Table 1 shows the source files created or heavily altered and their function. Appropriate header files ware also constructed.

```
if(Rank_MPI){ //-Only for root
       DataSendAnyMPI mpisend(Rank_MPI); //-Create buffer for sending data
       unsigned sbegin=0;
       for(int c=0;c<Size_MPI;c++){ //-Send information to all nodes.
          if(c){
            mpisend.AddDataUint(nslices); //-Pack unsigned integer to buffer
            mpisend.Send(TAG_EXAMPLE,c,Log_MPI); //-Send data with tag
              }
          else NumSlices=nslice; //-Store data for root
       }
}
else{ //-Not the root process
       DataRecvFixedMPI mpirecv(Rank_MPI,0); //-Buffer to receive data
       mpirecv.Recv(TAG_EXAMPLE,Log_MPI); //-Receive tagged data
       NumSlices=mpirecv.UnpackDataUint(); //-Unpack unsigned integer
}
```

Figure 3: Example of send-receive data process within DualSPHysics.

Filename	Function
BalanceMPI.cpp	Load Balancing using 1D decomposition algorithm
BufferMPI.cpp	Functions to assist with sending and receiving buffer data
CellDivCpuMPI.cpp (replaces JCellDivCpuSingle.cpp)	Cell division, particle reordering and neighbour list creation

DataBuffMPI.cpp	Functions for buffer creation and data handling
DataCommMPI.cpp	Implements the necessary functions for data transfer between MPI processes
HostMPI.cpp	Obtains information on the available nodes for the MPI process
InfoMPI.cpp	Implements class InfoMPI for file HostMPI.cpp
ObjectMPI.cpp	Handles tags for sending and receiving data
ParticleLoadMPI.cpp (replaces JPartsload4.cpp)	Loads data from input files
SliceMPI.cpp	Creates slices for the 1D decomposition algorithm. Handles initial decomposition.
SphCpuMPI.cpp (replaces JSphCpuSingle.cpp)	Main DualSPHysics file, oversees all operations. Also integrates the Zoltan library.
SphHaloMPI.cpp	Identifies particles belonging to the halo and exchanges data
SphMPI.cpp	Contains helper functions for SphCpuMPI.cpp. Also implements particle exchange.

Table 1: Function of new files for the integration of MPI within DualSPHysics

Spatial decomposition algorithms

The efficiency of a parallel code greatly depends on maintaining a similar workload across nodes. This is achieved by assigning different parts of the domain (referred to as sub-domains) to different nodes. The simplest such algorithm is one dimensional: the domain is divided across its largest physical dimension in parts with 2h width, referred here as slices [13]. Its implementation, along with the development of overlapping communications (covered in a later section) is the main objective of Workpackage 2 (WP2).

```
AllocSliceMemory(NParticles,NumSlices); //-Allocate slice memory based on the
number of particles and slices
if(NLocalParticles){ //-if particles have been loaded in the node from the
file, copy particle position data to prepare for decomposition struct
  memcpy(SlicePos,loadpart->GetPos(),sizeof(tdouble3)*NLocalParticles);
}
const unsigned cel;
for(unsigned p=0;p<ParticleCount;p++){ //-Loop over all local particles
  switch(DivAxis){ //-Identify decomposition axis
  //-Find the slice this particle should be sent
       case 0: cel=unsigned((SlicePos[p].x-MinPos.x)/CellSize); break; //-x
       case 1: cel=unsigned((SlicePos[p].y-MinPos.y)/CellSize); break; //-y
       case 2: cel=unsigned((SlicePos[p].z-MinPos.z)/CellSize); break; //-z
  }
  if(cel>=NumSlices)RunException(met,"Particles outside valid domain.");
  SlicePart[p]=cel; //-Store slice ID for the particle
  //-Store number of particles in the slice
```

```
   if(SliceIdp[p]<(NLocalParticles))Slice[cel]++;
 }
```

Figure 4: Algorithm for domain decomposition in slices.

The resulting slices are then assigned to the different nodes with each node having the same number of slices after the initial split. The slice of each particle (arrays temporarily loaded on host) is then identified (as shown in Figure 4) and its data are sent to the appropriate node. Data reordering and cell creation are then done in each node independently using the same algorithms as the original DualSPHysics code.

It is clear that in the majority of cases, nodes will not have an equivalent number of particles, as the number of particles in each slice will be different. To balance the load in each node, slices can be moved from one node to the other. This load balancing method, however, cannot be used for larger cases as it does not provide a fine level of control for cases where the particle distribution varies across the domain. This is particularly important for the free surface cases this project is aimed at simulating. A better indicator for this case would be retaining a similar number of particles across nodes as their number is directly linked to the number of neighbours and therefore to the number of computations [4].

This can be achieved by using a multidimensional spatial decomposition and load balancing method such as the Hilbert Space Filling Curve (HSFC). The HSFC is a variation of the Peano curve which maps all points in a 2-D or 3-D (or higher) area along a continuous curve. This method has already been used for SPH in previous eCSE reports for an incompressible SPH code [5]. Its implementation in DualSPHysics is the main focus of Workpackage 3 (WP3).

The points to create the HSFC mapping for DualSPHysics will be the cells of size 2h instead of the SPH particles. This will allow us to reuse the optimised CLL system to map the particles and facilitate identify neighbours without creating a global cell list. The static position of the cells, unlike the particles' rapid movement is also an advantage.

This algorithm will be used multiple times over the course of the simulation to account for fluid movement and differences on node workload. This will also make use of the cells: a new array *CellWeight* will identify the ratio of particles in each cell over the total amount of particles.

$$CellWeight\,(p) = \frac{N_{pc}}{N_t} \tag{7}$$

where N_{pc} is the number of particles in the cell and N_t is the number of particles in the simulation. Node workloads can then be identified by summing *CellWeight* for all local cells; if the imbalance among loads increases (greater than 20%) the load balancing algorithm will then be applied.

The HSFC algorithm is not be newly coded. Instead, the existing Zoltan library is integrated into DualSPHysics. This library is used for the development and optimization of parallel, unstructured and adaptive codes and is scalable for up to 10^6 cores. It also includes an unstructured communication package (Zoltan_Migrate) and a distributed directory algorithm to handle large data. Neither of these are currently in use but they should prove useful tools as the number of cells increases.

```
//-Declare Zoltan structures
struct Zoltan_Struct *cell_Z;
Zoltan_Data DataZ;
//-Load particle data from input files and perform an initial decomposition
using slices
Load_Particle_Data();
Initial_Slice_Decomposition();
Initial_Cell_Divide();
//-Get Cell id, node data and cell weights
CellDiv_MPI->GetLocalCellID(DomCells);
CellDiv_MPI->GetCellNodeList(AxisDiv,MinPos,DomainCells,Rank_MPI);
CellDiv_MPI->GetCellWeights(NumLocalCells,NCells);
//-Set Zoltan parameters and create data structure for Zoltan
cell_Z=Zoltan_Create(MPI_COMM_WORLD);
Set_Zoltan_Parameters();
DataZ=Set_Zoltan_Variables;
//-Set necessary variables for load balancing
Zoltan_Set_Num_Obj_Fn(cell_Z, Get_Cell_Number, &DataZ); //-Number of cells
Zoltan_Set_Obj_List_Fn(cell_Z, Get_Cell_List, &DataZ); //-List of cells
Zoltan_Set_Num_Geom_Fn(cell_Z, Get_Num_CellCoord, &DataZ); //-Number of
coordinates (2 for 2D and 3 for 3D)
Zoltan_Set_Geom_Multi_Fn(cell_Z, Get_CellCoord, &DataZ); //-Cell coordinates
Zoltan_Set_Obj_Size_Fn(cell_Z, Get_DataSize,&DataZ); //-Data size
Zoltan_Set_Pack_Obj_Fn(cell_Z, Pack_Data,&DataZ); //-Zoltan packing algorithm
Zoltan_Set_Unpack_Obj_Fn(cell_Z, Unpack_Data,&DataZ); //-Unpacking algorithm
```

Figure 5: Zoltan setup structure

```
//-Main load balancing function
errcode = Zoltan_LB_Partition(cell_Z, //-input(all other fields are output)
        &ChangePartitionZ,//-1 if partitioning was changed, 0 otherwise
        &Num_GID_Entries,//-Number of integers used for a global ID
        &Num_LID_Entries,//-Number of integers used for a local ID
        &Num_Import, //-Number of cells to be sent to this node */
        &Import_Global_GIDs,//-Global Cell IDs to be sent to this node
        &Import_Local_GIDs, //-Local Cell IDs to be sent to this node
        &Import_Procs,//-Source of each imported cell
        &Import_To_Part,//-New partition for each incoming cell (not used)
        &Num_Export,//-Number of cells sent to other nodes
        &Export_Global_GIDs,//-Global Cell IDs to be sent to other nodes
        &Export_Local_GIDs,//-Local Cell IDs to be sent to other nodes
        &Export_Procs,//- Nodes exported cells are sent to
        &Export_To_Part);//-Partition for exported cells(not used)
if (errcode != ZOLTAN_OK){
       MPI_Finalize();
       Zoltan_Destroy(&cell_Z);
       RunException(met,"Cell partition has failed");
}
NumLocalCells=NumLocalCells+Num_Import-Num_Export;//-New local cell count
//-Update arrays containing cell data
Update_LocalCellList(Num_Import,Import_Global_GIDs,
       Num_Export,Export_Global_IDs);
Update_CellNodeList(Num_Import,Import_Global_GIDs,Import_Procs,
        Num_Export,Export_Global_IDs,Export_Procs);
//-Identify particles that have been moved and send data to relevant node
Transfer_Particle_Data(CellNodeList,LocalCellList,Import_Procs,Export_Procs);
Cell_Divide();//-Reorder new cell data
```

Figure 6: Zoltan load balancing implementation

The integration of Zoltan is currently focused on the implementation of the HSFC. Figure 5 shows the setup algortim for using the Zoltan library while Figure 6 shows the setup procedure for the load balancing algorithm (which is only executed at the beginning of the simulation), while two new integer arrays are created to handle the HSFC data:

a) The *CellNodeList* array with size *Ncells* (total number of cells across domain) which will store the current node of each cell. It will be used for a more efficient identification of the neighbour cells. It is expected that as the number of cells increases, its efficiency will be decreased and the distributed datax directory functions will need to be used.

b) The *LocalCellList* array with size *NLocalCells* (number of cells on the node) which maps the cells used by the Zoltan library to the DualSPHysics *CellPart* array which holds the reordered particle data and the cells to which they have been assigned.

The use of an intermediate array (*LocalCellList*) between Zoltan and DualSPHysics is necessary due to the different size of the arrays used, as the internal. Its use could be avoided if particle reordering is altered to follow a Hilbert curve. The HSFC maintains spatial locality which is beneficial for interpolation methods, however, the current reordering system is the initial DualSPHysics algorithm through the *CellPart* array.

It should be noted that both decomposition algorithms currently apply to every cell in the domain. As mentioned, this is a major problem for the 1D method. For the HSFC algorithm, while it will assign empty cells to nodes with lower workload, this is still an insufficient use of resources. An improvement for further releases would be identifying void cells so they can be ignored.

Particle and halo exchange systems

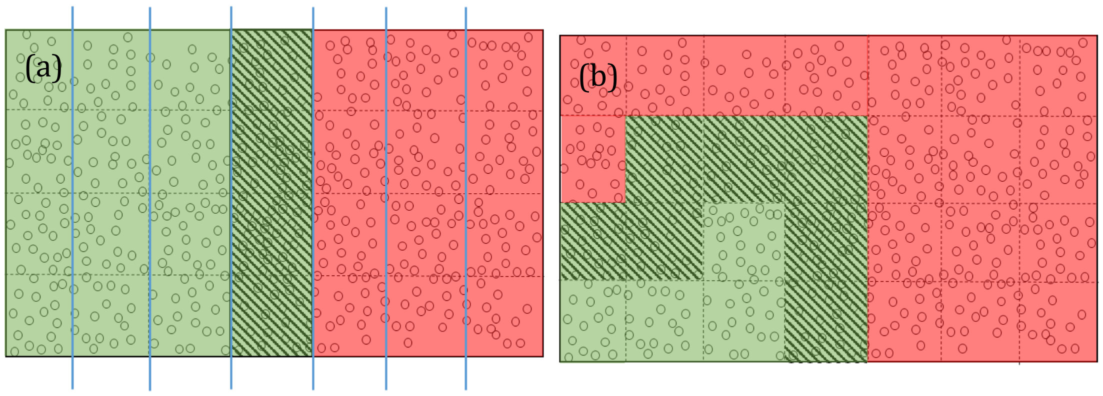


Figure 7: Border between two processes. The shaded area is the cells that need to be checked for the particle and halo exchange. Figure (a) shows the area required for the 1D domain decomposition and figure (b) shows the irregularly shaped subdomains created by the HSFC.

There are two major instances on which nodes need to exchange particle data:

(i) when a particle moves between nodes,
(ii) when the SPH interpolation requires particle data from other nodes.

These exchanges involve the edge of the sub-domain, the area within a distance $2h$ from the sub-domain boundary. This area is also referred to as the halo [13]. The $2h$ size is selected to be the same as the size of the cells; this guarantees that particles outside of the halo will not be involved in any exchanges as the adaptive time step criteria [3] ensure that changes in particle position are smaller than $2h$. An example of the cells and particles included in the halo can be seen in Figure 7 for both decomposition methods.

To address particles moving through nodes, a process referred heretofore as particle exchange, it is necessary to identify the dimensions of the sub-domain. For the slice method, this is relatively simple: the position of the slices is inherently known as they are of size $2h$ (one slice forms the edge). Therefore, by storing the minimum and maximum ID of the slices in the node (two integer numbers) it is possible to easily identify the sub-domain dimensions and compare them to the particle position.

Since the nodes created by the HSFC do not have a clear geometric shape, their basic unit, the cells (also of size $2h$) will be used to identify the sub-domain dimensions. A new array, *edgecell* is used to identify cells belonging in the edge of the sub-domain by checking whether its neighbours are in the *LocalCellList*. Particles in these cells are then checked against the cell dimensions and if they have moved out, their new cell (identified by the particle position) is checked against the *LocalCellList*. Data is then transferred to the new node if needed. At the time of writing this algorithm has not yet been completed.

```
 //-Send halo to neighbouring processes
ObjectMPI::StDatMPI_HaloParts HaloSend; //-Create packing array
ObjectMPI::StDatMPI_HaloParts HaloRecv; //-Create unpacking array

//-Identify and send cells and particles to particular processes
for (int iprocess; iprocess<NeighbourProcesses; iprocess++;){
       CellDiv_MPI->GetHalo(iprocess, EdgePart, CellNodeList, LocalCellList,
edgecell, HaloSend, Pos, VelRhop, IdPart);
       SendHalo->AddData(HaloSend[iprocess],
sizeof(ObjectMPI::StDatMPI_HaloParts));
       SendHalo->Isend(TAG_HALOPARTS,Log_MPI);
  }

//-Update Pressure Values for all particles
PreInteraction_Forces(tinter);

//-Compute forces for interior cells
Interaction_Forces(InPart, CellDiv_MPI->GetInCells(), CellDiv_MPI-
>GetCellDomainMin(), CellPart, Pos, VelRhop, IdPart, Press, Viscdt, DRho,
Accel);

//-Receive halo data and compute necessary forces
SphHalo* halo; //-Create structure to hold Halo data
```

```
for(int iprocess=0;iprocess<NeighbourProcesses;iprocess++){
      Recv->Recv(TAG_HALOPARTS,MPI_ANY_SOURCE,Log_MPI);
      const int halosender=Recv->GetSource();
//-Unpack Halo Data.
      Recv->UnpackData(&HaloRecv,sizeof(ObjectMPI::StDatMPI_HaloParts));
      const unsigned newnp=halo->GetNp(); //-New particles added
//-Adjust particle arrays with new data
      if((Np+newnp)>MaxParticles){ //-Increase allocated memory if needed
                MaxParticles+=newnp;
                ResizeCpuMemoryParticles(MaxParticles);
      }
      halo->ConfigParticleData(Np, EdgePart, HaloRecv, newnp, IdPart, Pos,
VelRhop);
//-Prepare halo cells for computation
      CellDiv_MPI->PrepareHaloCells(halo->GetCellDomainHalo(), halo-
>GetNcells(), halo->GetHaloCells(), LocalCellList, Np, newnp);
}

//-Compute forces for exterior cells
Interaction_HaloForces(EdgePart, CellDiv_MPI->GetOutcells(), CellDiv_MPI-
>GetCellDomainMin(), CellDiv_MPI->GetHaloCells(), CellPart, Pos, VelRhop,
IdPart, Press, Viscdt, DRho, Accel);
```

Figure 8: Code example showing the implementation of the halo exchange algorithm and overlapping communications

Apart from the particle exchange, data also have to be exchanged during the SPH interpolation when the neighbouring particles belong to another sub-domain. This is the case for all particles in the halo, whose neighbouring cells include the halo of the neighbouring sub-domains (as the halo size has been selected to be the same as a cell). This instance will be referred as the halo exchange [13] and requires transferring all the particle data within the halo. Figure 8 shows the implementation of the halo exchange algorithm within the force computation function.

For the one dimensional decomposition, the neighbouring nodes are inherently known, as are their minimum and maximum slice IDs (stored for the particle exchange). Therefore, identifying the relevant cells is not complicated. The halo exchange is more complicated for the HSFC, where neighbouring cells could be in multiple nodes. The use of the stored *CellNodeList* array however, allows us to identify the nodes the neighbouring cells belong to and transfer the relevant data through the *LocalCellList* array and the particle list. To reduce computations, only cells contained in the *edgecell* array will be considered. At the time of writing, this algorithm has not yet been completed.

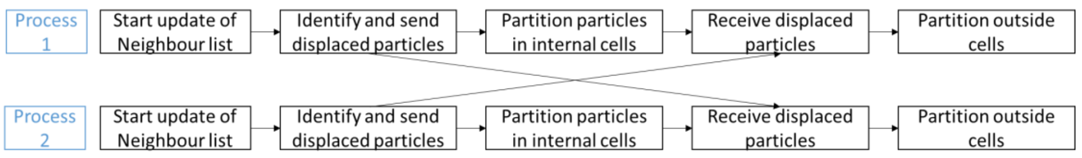

Process 1
Start update of Neighbour list
Identify and send displaced particles
Partition particles in internal cells
Receive displaced particles
Partition outside cells
Process 2
Start update of Neighbour list
Identify and send displaced particles
Partition particles in internal cells
Receive displaced particles
Partition outside cells

Figure 9: Overlapping communications for particle exchange.

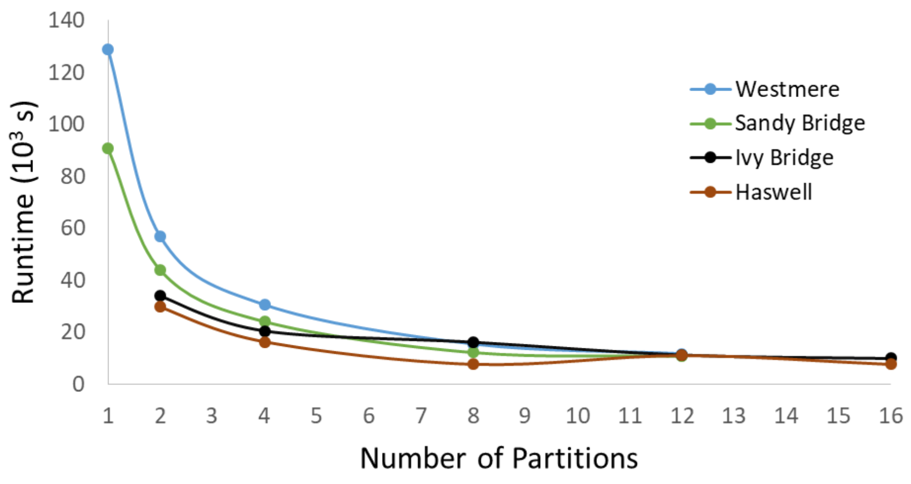
It is clear from the algorithms described above, that the particle and halo exchanges only affect data at the halo of the sub-domain. It is then possible to utilise asynchronous communications to overlap node communications with the particle computation minimising idle time. This can be done for both decomposition methods and is part of WP2. To achieve this, the particle list is divided in two arrays by identifying whether their current cell belongs to the halo or the inside of the domain by the processes (slice ID and the *edgecell* array) described above. The new arrays are referred to as *EdgePart* for the particles in the halo and *InPart* for the remainders.

For the particle exchange, particles in the *EdgePart* array are checked against the subdomain dimensions and any transfers are done using the data exchange algorithm shown in Figure 3. While the process waits to receive displaced particle data from others, the neighbour list for the inside particles can be updated. The received data is then added in the *EdgePart* array to update the neighbour list. A similar procedure can be followed for the halo exchange and SPH computation. Figure 9 shows the algorithm for particle exchange while Figure 8 shows the code dealing with halo exchange. The halo exchange follows the same steps except instead of displaced particles the data of the halo particles is transferred. This algorithm is not completed for the HSFC method and is not optimised for the 1D decomposition.

Results and discussion

The methods and algorithms described previously lead to the creation of two codes: a lightweight code suitable for execution in 100 cores with the one dimensional domain decomposition (Deliverable 1) and a more efficient code with the integration of the Zoltan library and the HSFC (Deliverable 2). Of particular interest for this report is the performance of the new codes.

(a)



(b)

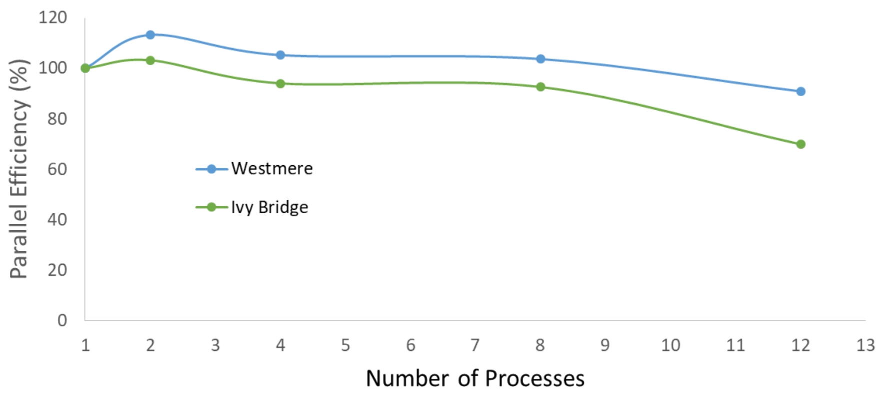


Figure 10: Runtime results and parallel efficiency compared to a single partition for the code using the 1D decomposition algorithm. Each partition is a single processor using 6 cores through OpenMP.

Westmere	Xeon X5650 - 2.66GHz (2x6-core)
Sandy Bridge	Xeon E5-2640 - 2.5GHz (2x6-core)
Ivy Bridge	Xeon E5-2650 v2 – 2.6GHz (2x8-core)
Haswell	Xeon E5-2690 v3 – 2.6GHz (2x12-core)

Table 2: CPU processors used for the runtime results.

For the first code, runtime results have been produced. A still-water case with 700,000 particles was used to minimise variation in the results. The case was executed with different processors to look at the effect of different hardware on the parallel efficiency and execution time of the simulation. Figure 10 shows the runtime results and the weak parallel efficiency, computed through Equation 9 for different processors, outlined in Table 2.

$$E_p = \frac{t_p}{Pt_1} 100\% \tag{8}$$

In Equation (9), E_p is the weak parallel efficiency, P is the number of processes, t_p is the runtime of an individual process in a simulation with P processors and t_1 is the runtime of a simulation with a single process. The results show a clear reduction in runtime when using multiple processes and the efficiency results show that the multiple processes are utilised close to their ideal capability. A small drop can be observed for the Ivy Bridge simulation when increasing the number of cores. An investigation identified that the reason is the use of the *MPI_Allreduce* function for identifying and transmitting the minimum time step to all processes.

For the code with the Zoltan library, since the implementation of the particle and halo exchange systems have not been completed it is not possible to produce runtime results. It is however, still possible to investigate the partition of the domain with the HSFC algorithm. The still-water case is not ideal here as the domain, filled with particles, is evenly divided by slices. Instead, a dam break case, where the majority of the domain is void will be used.

Figure 11 and Figure 12 show the decomposition of the dam break domain by both algorithms used. For clarity, only 8 processes have been used. The HSFC algorithm concentrates the processes to the cells containing the water particles with the void area handled by one or two processes due to the use of the *CellWeight* array for load balancing. The geometric decomposition of the slices, on the other hand uses three of the processes for the upper area of the domain, where there are no water particles at the moment. As a result, the available resources are much more evenly distributed for the HSFC algorithm.

Limitations and Incomplete Progress

The lightweight parallelised version of DualSPHysics has been completed and scales well up to 100 cores. In terms of other open-source SPH codes available, this does not represent a step forward in the state-of-the-art. It is disappointing to inform the funders that the time allocated to all the tasks originally planned for the entire project were clearly over optimistic. As mentioned throughout the text, the deliverables outlined for this project have not been completed. The main reason is that the complexity and time needed for the phases outlined in the proposal was underestimated. Phases 1 and 2 required 10 months to reach their current point with Deliverable 1 not being optimised, leaving insufficient time to complete Phase 3 and Deliverable 2. We estimate and additional 6 months would be needed to complete Phase 1-3. This means that the massively parallel code intended for use over 1000s of cores is not complete and does not run. Our intention is still to complete this work but it will have to be completed post project.

Dissemination

The work was presented at the 3rd International DualSPHysics Users Workshop in November 2017 with a positive reception.

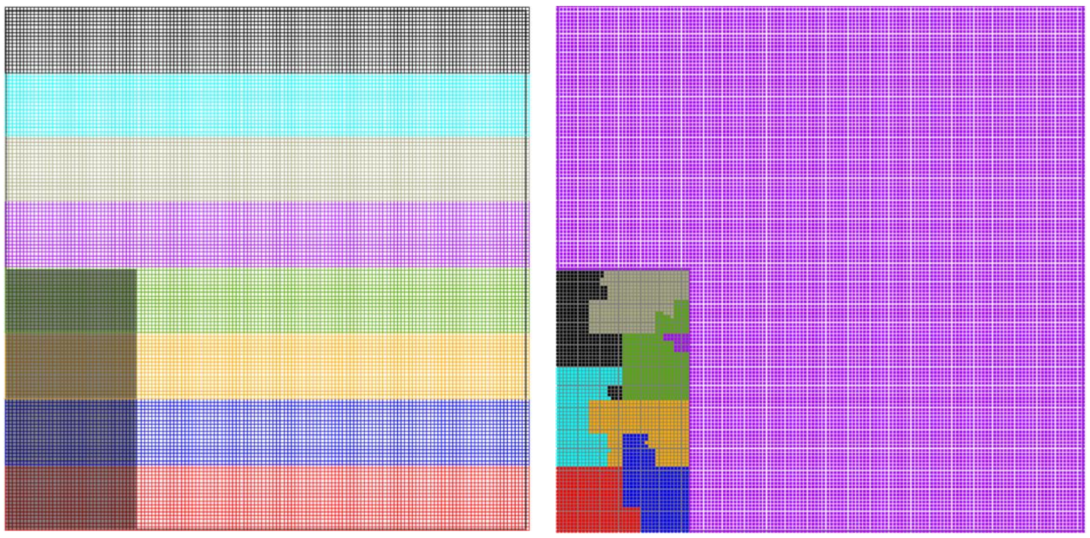


Figure 11: Cell decomposition of a 2D domain using the slice method (left) and the HSFC algorithm (right). The shaded area is the initial position of the water.

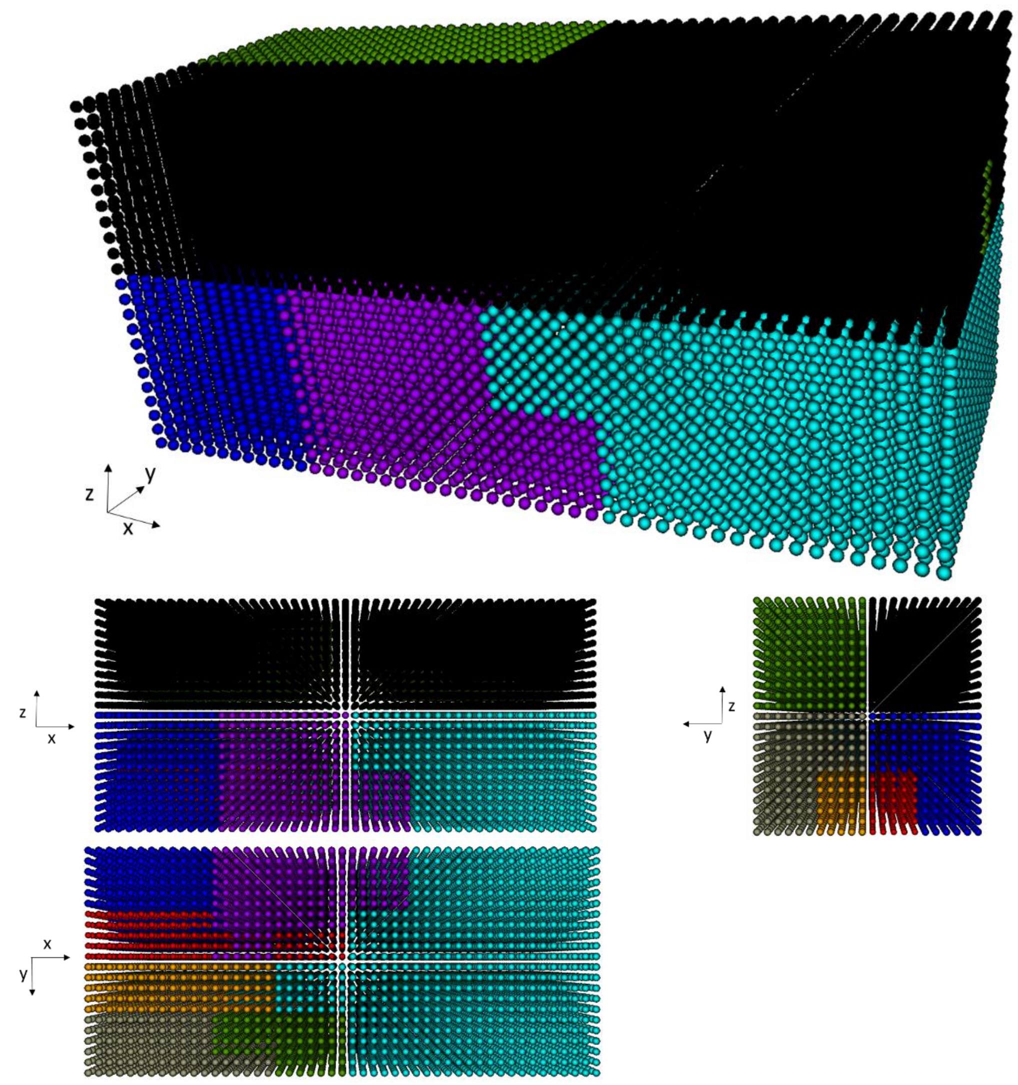


Figure 12: **Cell decomposition of a 3D domain using the HSFC algorithm**

[1] R. A. Gingold and J. J. Monaghan, "Smoothed Particle Hydrodynamics - Theory and Application to Non-Spherical Stars," *Monthly Notices of the Royal Astronomical Society,* vol. 181, pp. 375-389, 1977 1977.

[2] L. B. Lucy, "Numerical Approach to Testing of Fission Hypothesis," *Astronomical Journal,* vol. 82, pp. 1013-1024, 1977 1977.

[3] A. J. C. Crespo, J. M. Dominguez, B. D. Rogers, M. Gomez-Gesteira, S. Longshaw, R. Canelas*, et al.*, "DualSPHysics: Open-source parallel CFD solver based on Smoothed Particle Hydrodynamics (SPH)," *Computer Physics Communications,* vol. 187, pp. 204-216, Feb 2015.

[4] G. Oger, E. Jacquin, M. Doring, P. M. Guilcher, R. Dolbeau, P. L. Cabelguen*, et al.*, "Parallel hybrid CPU/GPU acceleration of the 3-D parallel code SPH-flow," presented at the 5th international SPHERIC Workshop, Manchester, UK, 2010.

[5] Guo, X., Rogers, B.D., Lind, S.J., Stansby, P.K., 2018, New massively parallel scheme for Incompressible Smoothed Particle Hydrodynamics (ISPH) for

highly nonlinear and distorted flow, *Computer Physics Communications*, in press, doi: 10.1016/j.cpc.2018.06.006.

[6] G. C. Fox, "A Graphical Approach to Load Balancing and Sparse Matrix Vector Multiplication on the Hypercube," New York, NY, 1988, pp. 37-61.

[7] D. Hilbert, "Über die stetige Abbildung einer Linie auf ein Flächenstück," *Mathematische Annalen,* vol. 38, pp. 459-460, 1869.

[8] J. J. Monaghan, "Smoothed particle hydrodynamics," *Reports on Progress in Physics,* vol. 68, pp. 1703-1759, Aug 2005.

[9] D. Violeau and B. D. Rogers, " Smoothed particle hydrodynamics (SPH) for free-surface flows: past, present and future," *Journal of Hydraulic Research,* vol. 54, pp. 1-26, 2016.

[10] A. Colagrossi and M. Landrini, "Numerical simulation of interfacial flows by smoothed particle hydrodynamics," *Journal of Computational Physics,* vol. 191, pp. 448-475, Nov 1 2003.

[11] J. M. Dominguez, A. J. C. Crespo, M. Gomez-Gesteira, and J. C. Marongiu, "Neighbour lists in smoothed particle hydrodynamics," *International Journal for Numerical Methods in Fluids,* vol. 67, pp. 2026-2042, Dec 30 2011.

[12] A. C. Crespo, J. M. Dominguez, A. Barreiro, M. Gomez-Gesteira, and B. D. Rogers, "GPUs, a new tool of acceleration in CFD: efficiency and reliability on smoothed particle hydrodynamics methods," *PLoS One,* vol. 6, p. e20685, Jun 13 2011.

[13] D. Valdez-Balderas, J. M. Dominguez, B. D. Rogers, and A. J. C. Crespo, "Towards accelerating smoothed particle hydrodynamics simulations for free-surface flows on multi-GPU clusters," *Journal of Parallel and Distributed Computing,* vol. 73, pp. 1483-1493, Nov 2013.