Blucher Mechanical Engineering Proceedings
May 2014, vol. 1 , num. 1
www.proceedings.blucher.com.br/evento/10wccm

10th World Congress on Computational Mechanics
8-13 July 2012 • São Paulo • Brazil

EXPLICIT MOVING PARTICLE SIMULATION METHOD ON GPU CLUSTERS

D. Taniguchi[1] , L. M. Sato[1], L. Y. Cheng[2]

[1]Department of Computer Engineering and Digital Systems, Escola Politécnica at the University of São Paulo (denis.taniguchi@usp.br, liria.sato@poli.usp.br)

[2]Department of Construction Engineering, Escola Politécnica at the University of São Paulo (cheng.yee@poli.usp.br)

Abstract. *Moving Particle Semi-implicit (MPS) is a Lagrangian, meshfree, computational method for fluid simulation. This work focus on using GPU clusters for MPS simulations. To accomplish this, we have to deal with two different levels of parallelism: one responsible for making different cluster nodes work together in a distributed memory system, and the other using the parallelism of GPU devices available on each node. First we present a performance comparison between single-node GPU and single-node multithreaded CPU implementations to investigate GPU speedups. Further, we analyze the performance in a multi-node GPU cluster environment.*

Keywords: *Particle Method, Computational Fluid Dynamics, High Performance Computing, GPU, CUDA.*

1. INTRODUCTION

MPS method was originally introduced in [10] as a Lagrangian, meshfree method for simulation of incompressible fluids. The fluid is represented with a finite number of particles that can freely move in space. The particle behavior is affected only by the surrounding particles, the closer the other particles are, the greater the influence. In summary, the fluid is represented as freely moving particles without the use of a grid or mesh, and the differential operators of the governing equations are replaced by using Discrete Differential Operator on Irregular Nodes (DDIN) [9], and solved by a semi-implicit algorithm.

The unique computational aspect of the method is an NxN sparse linear system, with N being the number of particles, which need to be solved at each time step to guarantee incompressibility.

To avoid the overheads of solving a large sparse linear system, [17] proposes a weakly compressible MPS method. A linear system to solve the pressure Poisson equation, which will be responsible for maintaining the constant density within the fluid, was replaced by an equation of state [4] commonly used in SPH method [14]. This equation relates density with

pressure in a very simple way, but incompressibility is not guaranteed. On the other hand the method becomes fully explicit and well suited for computation on GPUs.

To allow the computation in a distributed memory system we must perform domain decomposition. Each sub-domain is simulated by a separate process and communication between them must be minimized. Moreover, load balance must be a concern to avoid idle processes during the simulation. The ideal scenario is to have all the processes containing the same amount of particles throughout the simulation.

Computations on GPUs have been the focus of a great deal of research over the past few years, with speedups increasing ten or even hundreds of times than when they were implemented on CPUs [1][5][6][11]. Frameworks have been released to accelerate software development [13][15], in an effort to make the burden of re-writing any software to work on GPUs more bearable. Despite the speedups rates, and attractive GFLOPS/dollar ratios, not all applications can benefit from GPU computing. Implementations of GPU kernels usually need a careful study on how to efficiently employ each available resource, and successfully hide latencies.

This work is focused on the use of GPU clusters for MPS simulations. Two levels of parallelism are implemented: one by dividing the simulation domain among cluster nodes and using Message Passing Interface (MPI) for communication between sub-domains, and another through the use of GPU devices in the computation of the explicit MPS method. With this development we expect to be capable of simulating large amounts of particles and expand the possibilities for applications of particle methods in engineering analysis.

Section 2 investigate the MPS method and related computer algorithms are subject of the following Section 3. Section 4 depicts how parallelization using GPUs was performed followed by a description of the parallelization on distributed memory systems on Section 5. Results are presented on Section 6 along with a discussion concerning the implementation efficiency and computation speedups.

2. MPS METHOD

The MPS method is based on the discretization of the fluid volume in a finite number of particles. The contribution of a particle i to a particle j, and vice versa, is defined by a kernel function defined as:

$$w(r_{ij}, r_e) = \begin{cases} \frac{r_e}{r_{ij}} - 1, r_{ij} < r_e \\ 0, r_{ii} \geq r_e \end{cases} \quad (1)$$

where r_{ij} is the distance between the particles and r_e the radius of influence. The kernel function interpolates the physical quantities of a region working as a smoothing function of the surrounding particles.

A dimensionless quantity called particle number density is written as:

$$\langle n \rangle_i = \sum_{i \neq j} w(r_{ij}, r_e) \tag{2}$$

and is closely related to the local density of the fluid. The operator $\langle \quad \rangle$ is the kernel weighting operator, which means that the quantity is a weight average using the kernel function over the vicinities of particle i. Computationally, it indicates that a summation of a certain expression relating particle i parameters, and each of its neighbors, j parameters, weighted by a kernel function, will be performed.

The kernel weighting operator is also used in the definition of the gradient vector:

$$\langle \nabla \cdot \phi \rangle_i = \frac{d}{n_0} \sum_{i \neq j} \left(\frac{\phi_j - \phi_i}{{r_{ij}}^2} (r_j - r_i) w(r_{ij}, r_e) \right) \tag{3}$$

and the Laplacian:

$$\langle \nabla^2 \phi \rangle_i = \frac{2d}{\lambda n_0} \sum_{i \neq j} \left((\phi_j - \phi_i) w(r_{ij}, r_e) \right) \tag{4}$$

in which, d is the number of space dimensions, n_0 is the constant particle number density, ϕ is a scalar quantity, and λ is a correction constant.

The calculation process has a prediction-correction integration method depicted in Figure 1.

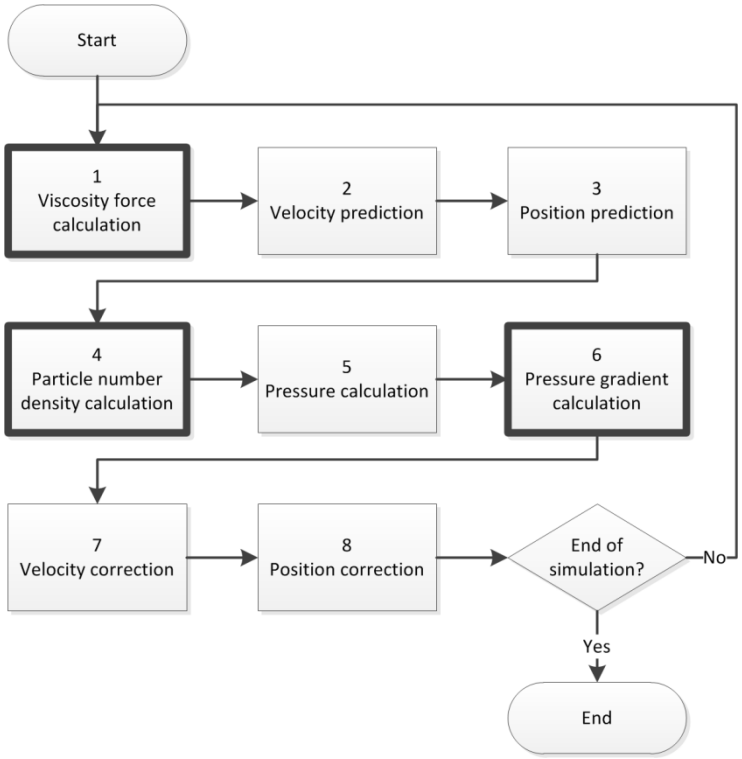


Figure 1. MPS method flowchart.

The first step is to calculate the viscosity force taking the Laplacian of the velocity, which involves a kernel weighting operator. Processes that require this operator are displayed with a thicker outline (processes 1, 4 and 6) and require a great computation effort. Velocity and position prediction are written as:

$$u_i{}^* = u_i{}^n + \Delta t f_i \tag{5}$$

$$r_i{}^* = r_i{}^n + \Delta t u_i{}^* \tag{6}$$

where $u_i{}^*$, $u_i{}^n$, Δt, f_i, $r_i{}^*$, and $r_i{}^n$ are the predicted velocity, velocity at step n, time step, force, predicted position, and position at step n, respectively. Particle number density is calculated according to (2).

Pressure is defined as:

$$p_i = {c_0}^2 \frac{\rho_0}{n_0}(n_i - n_0) \tag{7}$$

in which c_0 is the sound speed in the reference density ρ_0 [16].

Pressure gradient is calculated according to (3) for pressure. Velocity and position correction are given as follows:

$$u_i{}' = -\frac{\Delta t}{\rho_0}\nabla p^{n+1} \tag{8}$$

$$r_i{}' = \Delta t u_i{}' \tag{9}$$

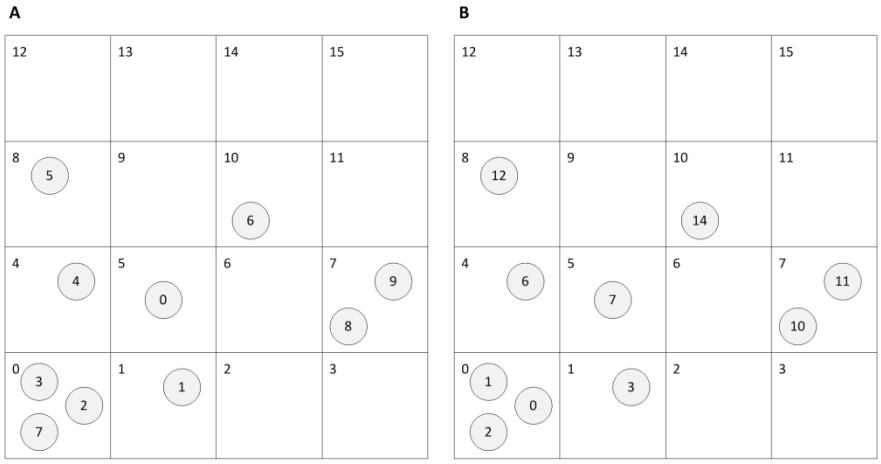
Velocity and position for the new time step is written as:

$$u_i{}^{n+1} = u_i{}^* + u_i{}' \tag{10}$$

$$r_i{}^{n+1} = r_i{}^* + r_i{}' \tag{11}$$

3. ALGORITHMS

This section describes the main algorithms used in the system implementation. First we delve into the kernel weighting operator and how particles are sorted in a grid to reduce its computational complexity. Second we look into how solid boundaries and free surfaces are handled. Third we describe the choice of a static domain decomposition, and how it affects load balance, followed by a explanation about the required communications steps using message passing between sub-domains. At last, we investigate how inflow and outflow particles can be effectively exchanged between sub-domains taking advantage of the fact that they are already sorted in a grid.



Index	0	1	2	3	4	5	6	7	8	9
x	1.5	1.7	0.8	0.2	0.7	0.2	2.3	0.3	3.2	3.7
y	1.5	0.7	0.5	0.8	1.7	2.8	2.2	0.2	1.2	1.7

Index	0	1	2	3	4	5	6	7	8	9
Cell	5	1	0	0	4	8	10	0	7	7

Index	0	1	2	3	4	5	6	7	8	9
Cell	0	0	0	1	4	5	7	7	8	10
x	0.8	0.2	0.3	1.7	0.7	1.5	3.2	3.7	0.2	2.3
y	0.5	0.8	0.2	0.7	1.7	1.5	1.2	1.7	2.8	2.2

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
First	0	3	4	4	4	5	6	6	8	9	9	10	10	10	10	10
Last	3	4	4	4	5	6	6	8	9	9	10	10	10	10	10	10
Size	3	1	0	0	1	1	0	2	1	0	1	0	0	0	0	0

Figure 2. Sorting algorithm.

3.1. Kernel weighting operator

Processes including the kernel weighting operator have a heavy computational load, so an optimized way to sum the contributions of each neighboring particle must be found. Fortunately this has been the subject of previous research [7][12], and a brief description is given below.

If we take the naïve approach considering every particle in the domain as a neighbor of another particle, we would end up with a $O(N^2)$ problem, with N being the total number of particles in the domain. This can be overcome by narrowing down the number of particles that we consider neighbors. Green [7] achieved that by sorting the particles in a grid. Figure 2 shows two grids, A and B, with A containing a random unsorted distribution of particles, and B being the result after sorting A. The first table shows two arrays containing the x and y positions of each particle.

The sorting algorithm consists of the following steps:

1. compute the cell index of each particle;
2. sort the particles by cell indices, also swapping the particle positions during the process;
3. find out the beginning and end (one past the end) index of each cell, and the number of particles lying on each cell.

The output arrays of each step are shown in Figure 2. The particle number density of particle i can be calculated using the sorted arrays according to Algorithm 1 presented below:

This approach reduces the order of complexity of processes that include the kernel smooth operator from $O(N^2)$ to $O(N)$, making it feasible computationally.

Algorithm 1

```
ParticleNumberDensity(i, x[], y[], cell[], first[], last[])
// i      particle index
// x      array of x positions
// y      array of y positions
// cell   array of cell indices
// first  array of  indices pointing to the beginning of each cell
// last   array of indices pointing to the end of each cell
1  result = 0
2  for c in NeighborCells(cell[i])
3  for j from first[c] to last[c]
4  dx = x[j] – x[i]
5  dy = y[j] – y[i]
6  r = sqrt(dx*dx + dy*dy)
7  if r < re
8  result += re/r - 1
9  return result
```

3.2. Boundary conditions

Solid boundaries are modeled as particles, making its implementation simple, but with the disadvantage of increasing the required number of particles [10][17]. There is also the need of introducing ghost particles to avoid drop of particle number densities near boundaries, which would make particles in those regions to behave wrongly and possibly pass through boundaries. Figure 3 shows an example of a fluid/boundary interface and the distribution of the different types of particle.

Therefore, an additional “type” attribute to each particle is required to define whether it is a fluid, boundary, or ghost particle as each type participates differently in the processes of Figure 1. For instance, during viscosity force calculation, the calculation must be performed for all fluid particles, considering their interaction with all but ghost particles.

Free surfaces are detected according to the following expression:

$$\langle n \rangle_i \leq n_0 \beta \tag{12}$$

in which β is a threshold coefficient. We used a β of 0.97 for our simulations and atmospheric pressure is imposed on free surface particles.

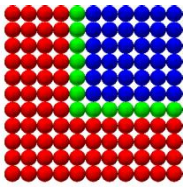


Figure 3. Fluid, solid boundary and ghost particles (colored in blue, green and red respectively).

3.3. Domain decomposition and load balance

Simulating particles on distributed memory systems inevitably scatters the particles in different processes in order to divide the work load, and requires some sort of communication between them to make the simulation occur as if it is all done in a single process.

Many previous researches addressed the problem of domain decomposition for particle methods [2][3][8]. Actually, a static partitioning scheme can work in the majority of applications such as sloshing, slamming, green water problems and simulation of a towing tank, to mention but a few, because the spatial behavior of the fluid is already known in some extent. The case chosen for this study was a 3D dam break problem. Figure 4 (screenshot from Paraview visualization software) shows the partitioning scheme that was used, dividing the domain in 4 parts aligned in the Z direction. Since it is capable of maintaining the same amount of particles in each sub-domain throughout the simulation because there is almost no inflow or outflow particles in Z direction, the partitioning is kept unchanged maintaining a good load balance.

Communication must occur to provide information of neighboring particles lying in neighboring sub-domains. For a given particle within the boundary of sub-domain A there are neighbor particles residing in B, a sub-domain that must be considered while calculating, for example, its particle number density. Hence, B must communicate the position of those par-

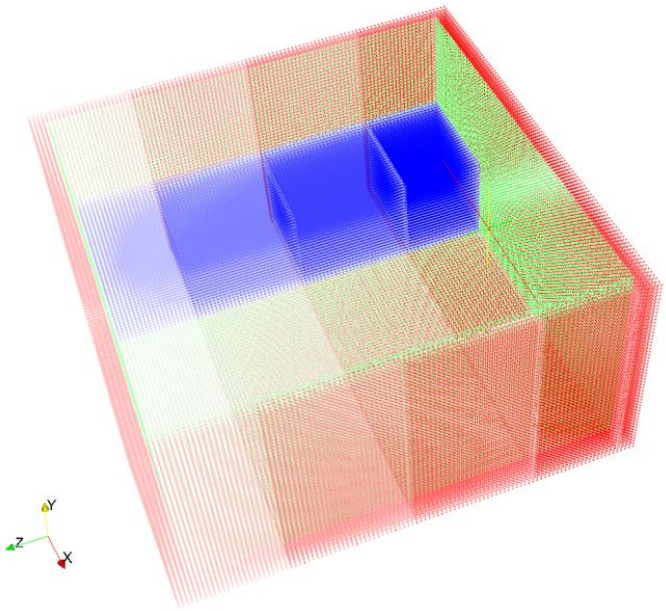


Figure 4. Partitioning scheme used for the 3D dam break case.

ticles to A before the particle number density can be calculated. Furthermore, to minimize communication, only particles close to the boundary should be transferred. Since all particles are kept sorted in a grid, only particles inside boundary cells are exchanged. Each process responsible for performing the calculations of a sub-domain keeps an updated list of boundary cell indices for each neighbor sub-domain.

3.4. Communication

Communication must occur between neighbor sub-domains. For a given sub-domain, the number of neighbors varies according to the domain partitioning scheme. For simplicity, we considered a two dimensional case with two sub-domains A and B, each divided by a 2x2 grid presented in Figure 5. We call the extended grid of A the 4x4 cells composed by the union of the 2x2 white cells and the surrounding hatched cells. Sorting is made using the extended grid, so the number of cells in A is 16. The same applies for sub-domain B, although its extended grid is not shown. Tables I, II, III and IV are the cell ranges for sub-domain A, particle positions and cell indices for sub-domain A, cell ranges for sub-domain B, and particle positions and cell indices for sub-domain B, respectively.

The algorithm to send the particle positions of sub-domain B boundary cells to sub-domain A consists of the following steps:

1. Test the intersection of sub-domain B grid with sub-domain A extended grid, finding which cells are in the intersection region, in this case cell 5 and 9.
2. Copy to a container (Table VI) the ranges of positions for cell 5 and 9 given by Table IV, and keep track of cell ranges using another container (Table VII).
3. Send Table VI and VII to sub-domain A, appending the positions to the end of the container holding the particle positions of A (Table V). Update the cell ranges of A (Table VIII) to point the correct indices of Table V for the inbound particles of cell 7 and 11.

This procedure avoid sorting the inbound particles taking advantage of the fact that they are already sorted in the communicating sub-domain, by providing additional information about the cell ranges (Table VII) which does not have a significant impact in communication and computation. The other advantage is the clear separation in Table V of particles lying in the sub-domain (inner particles) and the ones received from neighbor sub-domains (inbound

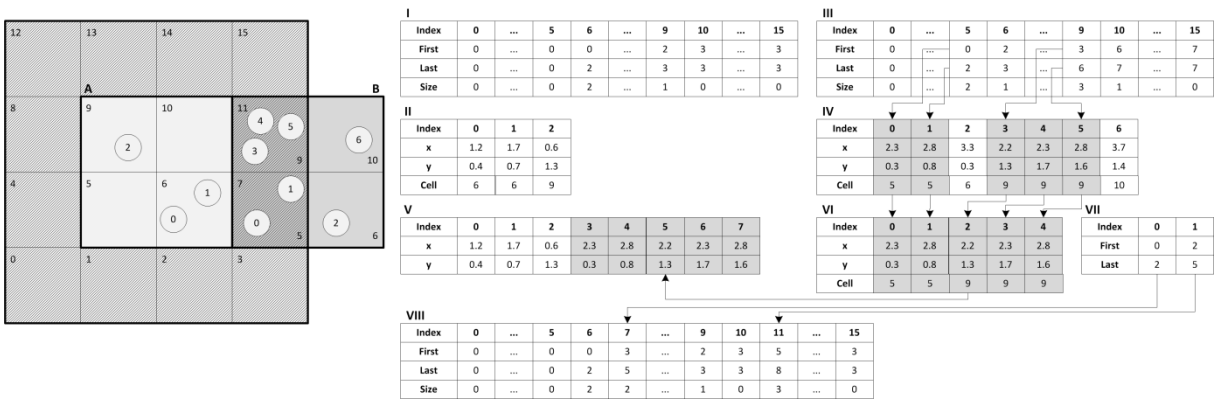


Figure 5. Algorithm for communicating particles between sub-domains.

particles). When calculations need to be carry out only for inner particles, it can be done just by adjusting the computation range, which is put into practice in process 7 and 8 of Figure 1 saving computational time.

Figure 5 shows a more detailed flowchart for the case of distributed memory systems. The exact attributes that are exchanged in each communication step are as follows:

I. position, velocity, type and cell ranges;
II. viscosity force;
III. particle number density;
IV. position.

Notice that the cell ranges are communicated only once, as they are not subject for change inside one time step. At first, communication IV didn't seem to be necessary, since all the information required to perform the sorting process in already locally available. Nevertheless, slight fluctuations in computation between GPUs was affecting the detection of inflow and outflow particles. This step ensures that for a given particle every sub-domain has the same position value, and inflow/outflow detection will be handled consistently among them.

3.5. Inflow and outflow

Sub-domains must be capable to find out if a particle has left the sub-domain (outflow) or entering the sub-domain (inflow). Furthermore, particles must make the transition in a consistent manner to avoid redundancy among sub-domains or disappearance while crossing boundaries.

The fact that we already have the inbound particles attributes appended in the local containers worked in our favor as applying the sort algorithm, with a small modification in how cell indices are computed, can be sufficient to treat inflow/outflow particles. Cell indices are computed verifying if the particle is inside the inner grid. If not, a value of ε is given where ε is much greater than the maximum cell index. After sorting was performed, all particles outside the inner grid will be gathered at the end of the container. We look for the first particle with cell index equal to ε, and all particles that come before it are considered to belong to that sub-domain while all others are discarded.

For instance, consider that particle 4 of sub-domain B in Figure 5 moved toward sub-domain A and is making its transition to A's cell 10. Since it is already present in Table V, sorting A will swap its data to cell 10 making it an inflow particle in sub-domain A. On the other hand, the same particle will be outside the inner grid of B, allowing it to be discarded after performing the sort process in B, distinguishing it as an outflow particle in sub-domain B. On the other hand, if the corrected position of particle 4 computed in A is slightly different from the one computed in B, an inconsistent state may be reached, since particle 4 may be seen as an inflow particle in A, but not as an outflow particle in B, for example. That would originate a duplicate particle in A after B communicated its particles (Communication I) in the following time step. As pointed out before, that is why Communication IV was needed, given the fact that slight differences in computation do occur between GPUs devices

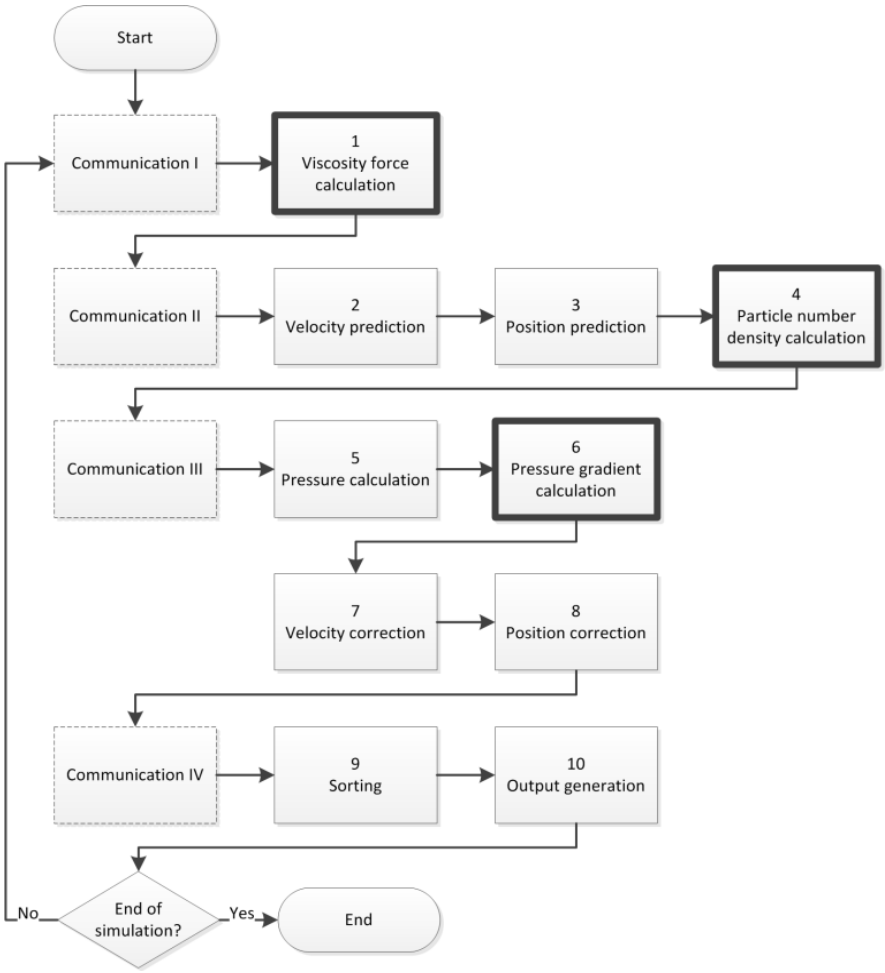


Figure 6. Flowchart for distributed memory systems.

4. COMPUTING ON GPU

GPU computing was implemented using NVIDIA's CUDA framework and since the release of version 4.0 of the toolkit, it includes a library namely Thrust which is based on C++ Standard Template Library (STL) and Boost's Iterators library. It features 2 special containers for different memory spaces, i.e. in host memory or device memory, and data transference between spaces. Algorithms can operate on containers through the use of iterators, and the library automatically executes on GPU or CPU depending on the data memory space. It is also possible to use an OpenMP backend and run parallel on CPU.

Processes 2, 3, 5, 7, and 8 in Figure 1 are simple iterations over all particles which enables a straightforward utilization of the featured "transform" algorithm. Processes 1, 4, and 6 required an implementation similar to Algorithm 1 for both CUDA and OpenMP backend.

Coalesced global memory access was guaranteed by using separate containers for each particle attribute.

Concurrent execution between host and device could be sought during communication. More specifically, processes 2, 3, 5, 7 and 8 could perform their computation for inner particles, while inbound particles are been communicated. Unfortunately this would require the use multiple CUDA streams, currently not supported by Thrust.

Texture memory was used for cell range data that can optimize the cost of memory reads only on a cache hit in texture cache.

5. COMPUTING ON DISTRIBUTED MEMORY SYSTEMS

Message Passing Interface (MPI) was used for communication between cluster nodes. When computing using GPUs all data resides in device memory, but MPI requires data to be available in host memory. To avoid data transference overhead between device memory and host memory every time communication is needed we utilized a feature called page-locked memory, that maps host memory into the address space of the device.

Kernel launches are considered asynchronous in CUDA applications, meaning that control is returned to the host thread before their completion. We must ensure that all device tasks are completed before communication calls. This can be accomplished by using the function "cudaStreamSyncronize" before MPI calls with parameter set to the default stream.

MPI communication was implemented using non-blocking send and receive calls, MPI_Isend and MPI_Irevc respectively, followed by MPI_Waitall.

6. RESULTS

Results for 3D dam break simulations were obtained using two different clusters, with specifications shown in Table 1 and Table 2. The first goal was to compare the computing performance of a single multicore CPU with a single GPU device. We accomplished that by creating two different applications, namely MPS_OMP and MPS_CUDA, for parallelization using OpenMP and CUDA respectively. Since we used Thrust algorithms in the implementation of the MPS method, the source code for the two versions could be the same. Another goal was to check the performance improvements that could be reached by splitting the domain among different MPI processes, each one using one GPU device for the computation of the method. In summary, we built 3 different executables namely MPS_CUDA, MPS_OMP and MPS_CUDA_MPI, featuring single-node GPU, single-node multi-threaded CPU, and multi-node GPU execution schemes, respectively.

Table 1. GPU cluster configuration.

Processor	AMD Opteron 8384 2.7GHz
Cores per node	16
Memory	32GB
CUDA capable device	NVIDIA's Tesla S1070
Interconnection	Infiniband QDR 4x

Table 2. CPU cluster configuration

Processor	Intel Xeon X5560 2.8GHz
Cores per node	8
Memory	24GB
Interconnection	Infiniband QDR 4x

Table 3 presents the mean of the measured times, in seconds, of one time step of simulation for three 3D dam break cases, with different number of particles. The first three columns corresponds to the measurements of MPS_OMP running with 1, 4, and 8 threads in a multicore CPU. The forth column presents the times for MPS_CUDA running with a single GPU device. The fifth and sixth column are the times for MPS_CUDA_MPI using 4 and 9 processes respectively, with individual GPU devices. Figure 7 exhibits the obtained speedup rates for the case of approximately 700,000 particles if compared to MPS_OMP running with a single thread. We must be aware of the fact that the correct speedups would only be obtained by making the comparison against a pure serial implementation but we can assume that the overheads introduced by OpenMP parallelism are not significant and the numbers would be just slightly smaller.

Table 3. Mean computation time of one time step (seconds).

Particles	CPU 1 thread	CPU 4 threads	CPU 8 threads	GPU	GPU 4 processes	GPU 9 processes
141,168	1.097	0.389	0.206	0.101	-	-
452,528	3.973	1.671	0.888	0.282	-	-
698,578	7.467	2.628	1.389	0.513	0.163	0.092

MPS_CUDA is only 2.7 times faster than MPS_OMP with 8 threads for 698,578 particles. Profiling MPS_CUDA with Compute Visual Profiler, a profiling tool provided with CUDA Toolkit, reveled that around 95% of the computation time is spent in processes 1, 4, and 6 of Figure 1. Moreover it showed that they are compute bound, reaching a low occupancy of 41.6%. One way to overcome this would be to further divide the computation in simpler processes. That would require storing intermediate calculation values that lies inside the summation of the kernel weighting operators. However a great amount of additional memory

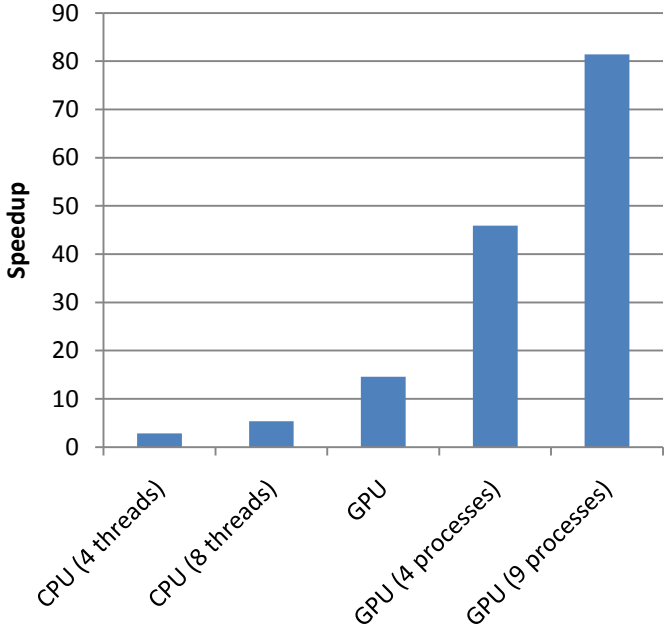


Figure 7. Speedup rates.

would be necessary since we would be storing one or more quantities for each pair of interacting particles. In addition, some extra computation would be required to allow data parallelism for interacting pair of particles.

We should also take into consideration that the GPU devices used to obtain the results are quite old, and given the fact that the implementation is currently compute bound, we should see some significant improvement if we execute in newer devices.

In this way, instead of trying to further optimize MPS_CUDA, we decided to concentrate our effort implementing MPS_CUDA_MPI, which seemed to have more immediate speedup opportunities.

Comparing the speedups rates achieved by MPS_CUDA_MPI with 4 and 9 processes reveals a good scalability. That means that communication is not the current bottleneck and the proposed algorithm introduced a very small overhead. However, we should remember that the partitioning scheme (Figure 4) minimizes communication since each sub-domain has only 2 neighbors. A different partitioning scheme, with more neighbors per sub-domain, would likely not scale that well.

In addition to reducing computation time, the domain decomposition allows the simulation of a great number of particles in a feasible time. A case with 1,190,678 particles was simulated in 4.5 hours using 8 processes and another with 7,329,376 particles in 17.5 hours using 14 processes.

7. CONCLUSION

We presented an algorithm to allow MPS simulations in distributed memory systems that greatly reduces the communication overhead. We also showed that MPS simulations can benefit from GPU computing, although speedup rates are not as high as achieved by other applications. To achieve higher speedup rates, optimized ways to perform the operations comprising the smoothing kernel operator need to be found. Possible alternatives include splitting the computation into simpler steps to reach higher device occupancy, or cleverly using the available resources of the GPU device that can impact performance such as shared memory.

Being able to compute in a GPU cluster further improved performance and allowed the simulation of a large number of particles.

8. REFERENCES

[1] Aji A. M., Zhang L., Feng W. C., “GPU-RMAP: accelerating short-read mapping on graphics processors”. *Proc. IEEE Conf. Computational Science and Engineering*, 168-175, 2011.

[2] Barnes J., Hut P., “A hierarchical O(nlogn) force-calculation algorithm”. *Nature*, 324, 446-449, 1986.

[3] Barnes J. E., “A modified tree code: don’t laugh; it runs”. *J. of Comput. Phys.*, 87, 161-170, 1990.

[4] Cole R., “Underwater explosions”. *Princeton University Press*, 1948.

[5] Friedrichs M. S., et al., “Accelerating molecular dynamic simulation on graphics processing units”. *J. Comput. Chem.*, 30, 864-872, 2009.
[6] Garland M., et al., “Parallel Computing Experiences with CUDA”. *IEEE Micro*, 28, 13-27, 2008.
[7] Green S., “Particle simulation using CUDA”. *NVIDIA Whitepaper*, 2010.
[8] Hamada T., Nitadori K., “190 TFlops astrophysical n-body simulation on a cluster of GPUs”. *Proc. IEEE Conf. for High Performance Computing*, 1-9, 2010.
[9] Isshiki H., “Discrete differential operators on irregular nodes (DDIN)”. *Int. J. Num. Meth. Eng.*, 88, 1323-1343, 2011.
[10] Koshizuka S., Oka Y., “Moving particle semi-implicit method for fragmentation of incompressible fluid”. *Nuclear Science and Engineering*, 123, 421-434, 1996.
[11] Lee M., Jeon J. H., Kim J., Song J., “Scalable and parallel implementation of a financial application on a GPU: with focus on out-of-core case.” *Proc. IEEE Conf. Computer and Information Technology*, 1323-1327, 2010.
[12] Le Grand S., “Broad-phase collision detection with CUDA”. *GPU Gems 3*, 697-721, 2008.
[13] Liu Y., Zhang E. Z., Shen X., “A cross-input adaptive framework for GPU program optimizations”. *Proc. IEEE Symp. Parallel & Distributed Processing*, 1-10, 2009.
[14] Monaghan J. J., “Simulating free surface flows with SPH”. *J. Comput. Phys.*, 110, 399-406, 1994.
[15] NVIDIA CUDA, *NVIDIA CUDA C Programming Guide*, 4.1, 2011.
[16] Oochi M., Yamada Y., Koshizuka S., Sakai M., “Explicit MPS algorithm for free surface flow analysis”. *Trans. of the Japan Society for Computational Engineering and Science*, 2011.
[17] Shakibaeinia A., Jin Y. C., “A weakly compressible MPS merhod for modeling of open-boundary free-surface flow”. *Int. J. Numer. Meth. Fluids*, 63, 1208-1232, 2010.