www.bsc.es

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

OpenMPSuperscalar: Task-Parallel Simulation and Visualization of Crowds with Several CPUs and GPUs

Hugo Pérez
UPC-BSC

Benjamin Hernandez
Oak Ridge National Lab

Isaac Rudomin
BSC

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

CUDA Center of Excellence

OAK RIDGE National Laboratory

OLCF Oak Ridge Leadership Computing Facility

March 2015

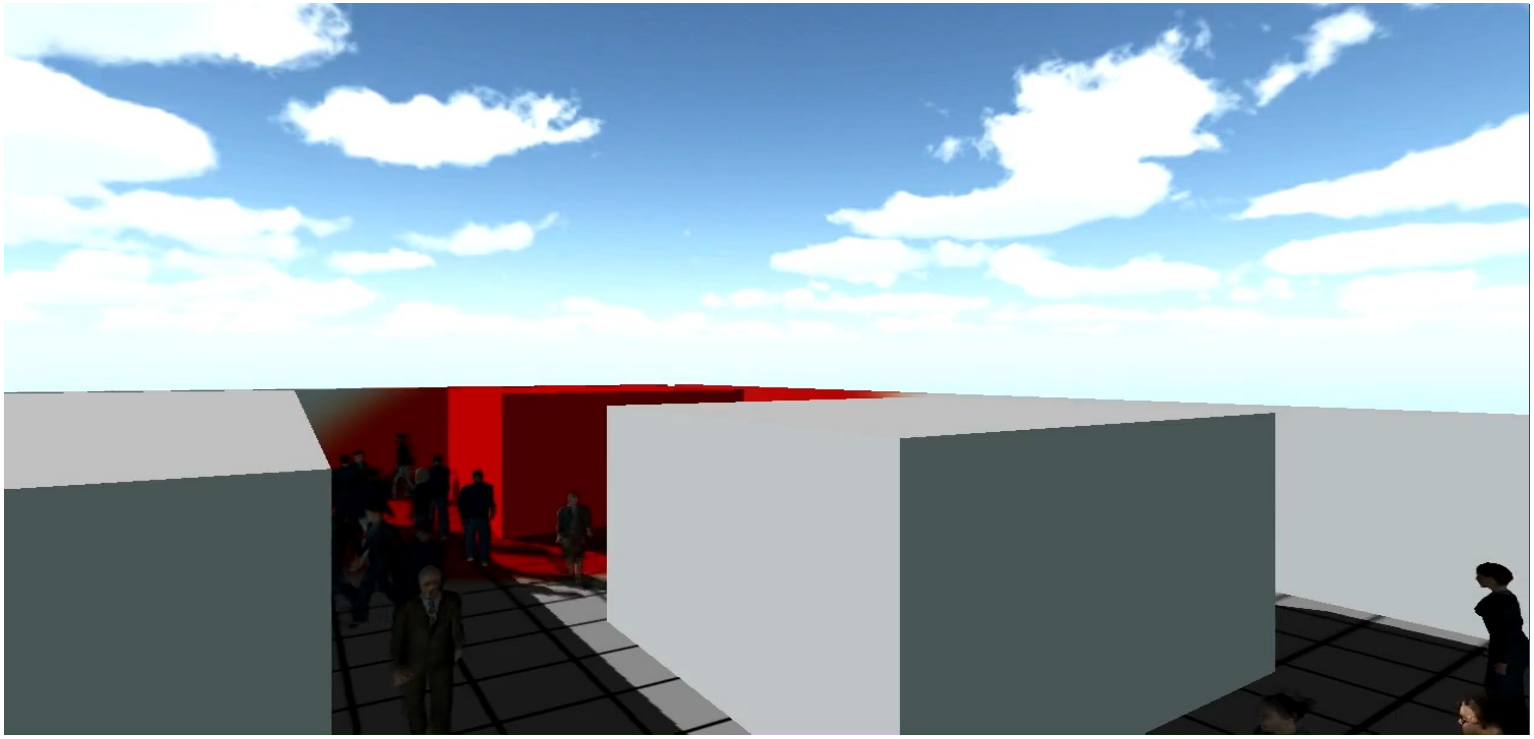
OUTLINE

- Introduction
- Algorithm
- CUDA version
- OpenMP Superscalar - OmpSs
- OmpSs version
- Results
- OmpSs and OpenGL
- Flexible Interactive Parallel Architecture for Visualization
- Conclusions
- Future Work

WHAT IS A CROWD SIMULATION?

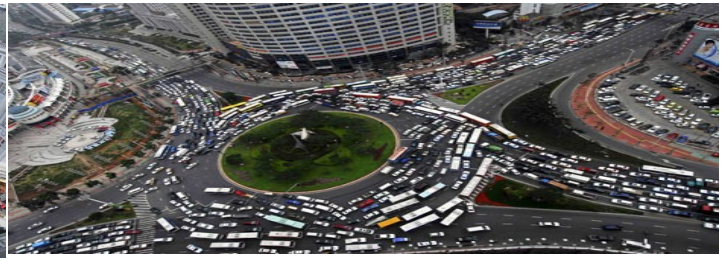
It is the process of simulate the behavior of a large number of characters.

Since each character takes decisions independently is a good candidate for parallel processing.



APPLICATIONS


Entertainment: Movies & Videogames

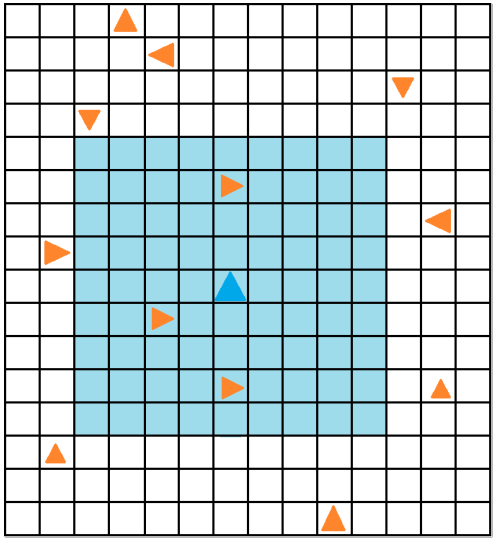



Urban Planning: Building construction & traffic routes


Training: Emergency Evacuation & Disaster Prevention

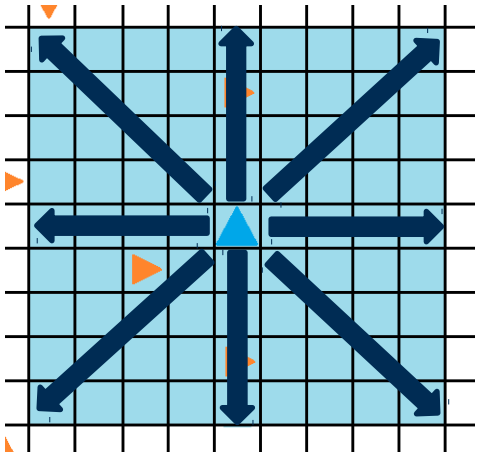
ALGORITHM

- Algorithm:
  - In our simulation wer represent the world (Navigation space) using a 2D grid



1. Initialize the agents with random values for: position, direction, and speed

2. For each agent:
   - Calculate collision avoidance
     - Check cells in counterclockwise in the eight directions within the given radius.
   - Update position and world cell status.
     - Move the agent in the direction in which there are more free cells.



ALGORITHM: MONOLITHIC WORLD

The main computation involves:

- Collision avoidance
- Update agents position & world status

We execute these operations in the GPU using CUDA.

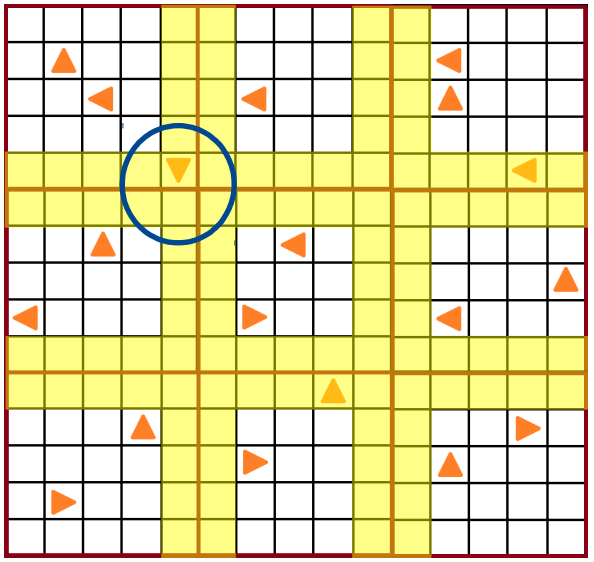
If we compute the data in a monolitic way:

- We transfer all the data to the GPU once, all iterations happens in the GPU, therefore we get significant speed up.
- It is simple
- The number of agents is limited for the size of the GPU memory.
- Works just for one GPU.

ALGORITHM: TILED WORLD

- For further scaling we divide the world in tiles

1. Set-up the communication topology between tiles, i.e. how they will interchange data
2. Initialize data
3. Exchange halos between neighbors tiles (yellow)
4. For each agent:
   - Compute collision avoidance
   - Update agent position and world cells status
5. Exchange agents



ALGORITHM: TILED WORLD 2

If we divide in tiles, the system has these characteristics:

- We can increase the number of agents in the simulation even using only one GPU, (the limit is the CPU memory)
- Using streams we can process different kernels in parallel
- We can overlap communication with computation
- We can use more than one GPU
- Implies new tasks

CUDA VERSION 1

Create a special structure to manage the data in the GPU

```
typedef struct {
        cudaStream_t      stream[MAX_NUM_BLOCKS];
        float4            *d_agents[MAX_NUM_BLOCKS];
        float4            *d_ids[MAX_NUM_BLOCKS];
        int               *d_world[MAX_NUM_BLOCKS];
} TGPUplan;
```

Allocate special host memory (pinned memory)

```
cudaMallocHost(...);
```

CUDA VERSION 2

```
int deviceCount;
cudaGetDeviceCount( &deviceCount ) ;

for (int i=0; i < deviceCount; i++ )
{
        cudaSetDevice(i);
        for (int j = 0; j < num_blocks/deviceCount; j++)
        {
         cudaStreamCreate( &plan[i].stream[j] );

         cudaMalloc((void**)&plan[i].d_agents[j], agents_total_buffer *sizeof(float4) );
         cudaMalloc((void**)&plan[i].d_ids[j], agents_total_buffer *sizeof(float4) );
         cudaMalloc((void**)&plan[i].d_world[j], world_cells_block *sizeof(int) );
        }
}
```

Split the data between the GPUs

Select the GPU in each operation

Create Streams

Allocate memory in the GPU

CUDA VERSION 3

```
for (int i = 0; i < deviceCount; i++){
  cudaSetDevice(i);
  for (int j = 0; j < num_blocks/deviceCount; j++) {
    int block = j + (i*num_blocks/deviceCount);

    //copying H2D
    cudaMemcpyAsync(plan[i].d_agents[j], agents[block], ... plan[i].stream[j]);
    cudaMemcpyAsync(plan[i].d_ids[j], ids[block], ... plan[i].stream[j]);
    cudaMemcpyAsync(plan[i].d_world[j], world[block], ... plan[i].stream[j]);

    refreshData(plan[i].d_agents[j], plan[i].d_ids[j], plan[i].d_world[j], ...
plan[i].stream[j]);

    //copying D2H
    cudaMemcpyAsync(agents[block], plan[i].d_agents[j], ... plan[i].stream[j]);
    cudaMemcpyAsync(ids[block], plan[i].d_ids[j], ... plan[i].stream[j]);
    cudaMemcpyAsync(world[block], plan[i].d_world[j], ... plan[i].stream[j]);  … }  }
```

Transfer data between the host and the GPUs

CUDA VERSION 4

```
for (int i=0; i < deviceCount; i++ )
{
    cudaSetDevice(i);
    for (int j = 0; j < num_blocks/deviceCount; j++)
        cudaStreamSynchronize( plan[i].stream[j] );
}
```

Synchronize operations

```
for (int i = 0; i < deviceCount; i++)
{
    cudaSetDevice(i);
    for (int j = 0; j < num_blocks/deviceCount; j++)
    {
        cudaFree(plan[i].d_agents[j]);
        cudaFree(plan[i].d_ids[j]);
        cudaFree(plan[i].d_world[j]);
        cudaStreamDestroy(plan[i].stream[j]);
    }
}
```

Release GPU resources

OPENMP SUPER SCALAR (OMPSS)

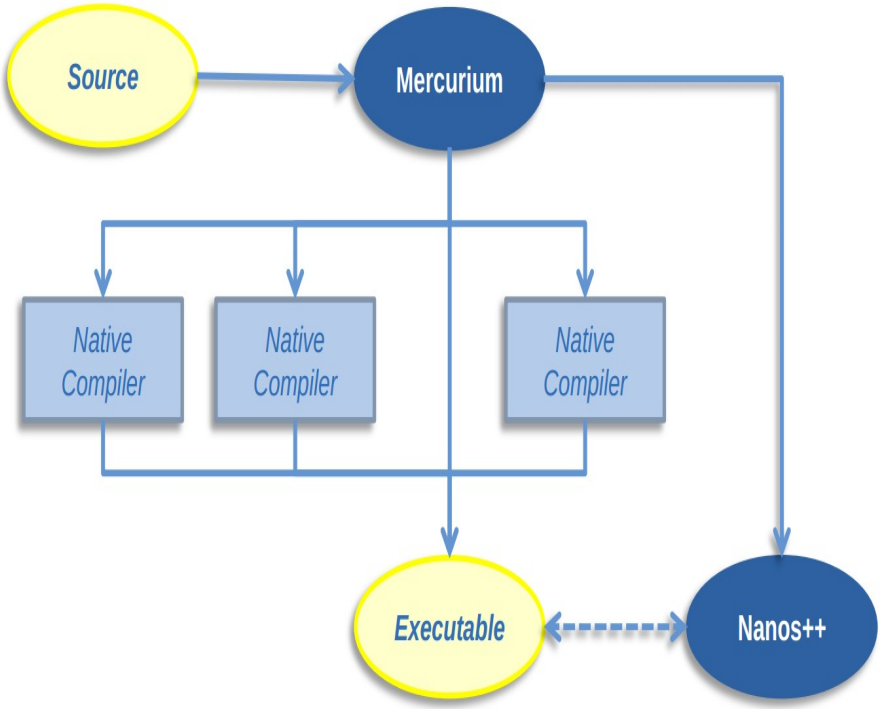
OmpSs does all these operations in an automatic way or some of them are not necessary.

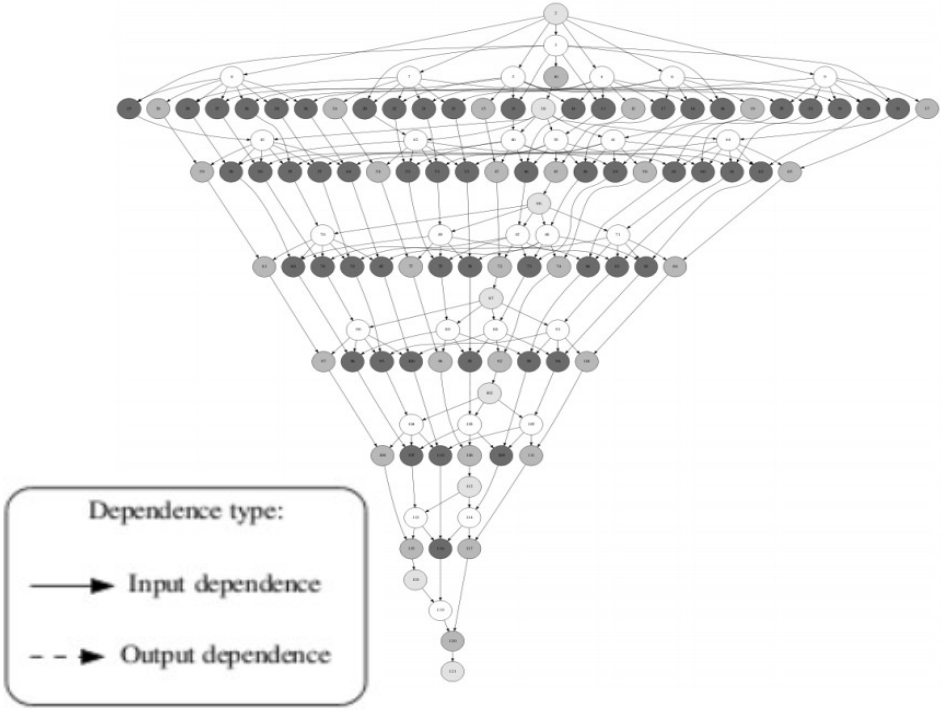
- You do not need to select the GPU
- You do not need to allocate special memory in the host (pinned memory) or declare different variables for host and GPUs
- You do not have to transfer data between the host and GPUs
- You do not need to create streams, Ompss implements them by default

OPENMP SUPER SCALAR (OMPSS) 2

OmpSs

- Allows us to perform tasks based on asynchronous parallelism.
- Its syntax is an extension of the directives used by OpenMP.
- Can be also applied to accelerators such as GPU.





Task Dependency Graph

OMPSS SYNTAX

```
#pragma omp task [ in(...)] [ out(...)] [ inout (...)]
{ function or code block }
```

Define the data dependences

Task implementation for a GPU device
The compiler parses CUDA or OpenCL kernel invocation syntax

```
#pragma omp target device ({ smp | cuda | opencl })     \
        [ndrange (…)]\
            [ implements ( function_name )]\
            { copy_deps | [ copy_in ( array_spec ,...)] [ copy_out (...)]
[ copy_inout (...)] }
```

Allows Multiple task implementations

Nanox++ ensures data is accessible in the device address space

```
#pragma omp taskwait [on (...)] [noflush]
```

Avoid copy of data

Tasks sync

Wait for specific variables

MINOTAURO

GPU Cluster.

128 Bullx B505 blades.

Hybrid architecture each node:

- **2x Intel Xeon** E5649 **6-Core** at 2,53 GHz
- 24 GB of RAM memory, 12MB of cache memory
- **2x NVIDIA M2090**, each one:
  - 512 CUDA Cores, 6GB of GDDR5 Memory.

CROWD DEFINITION OF TASKS 1

Definition of CPU Task

Maintain memory consistency only for a specific region of data

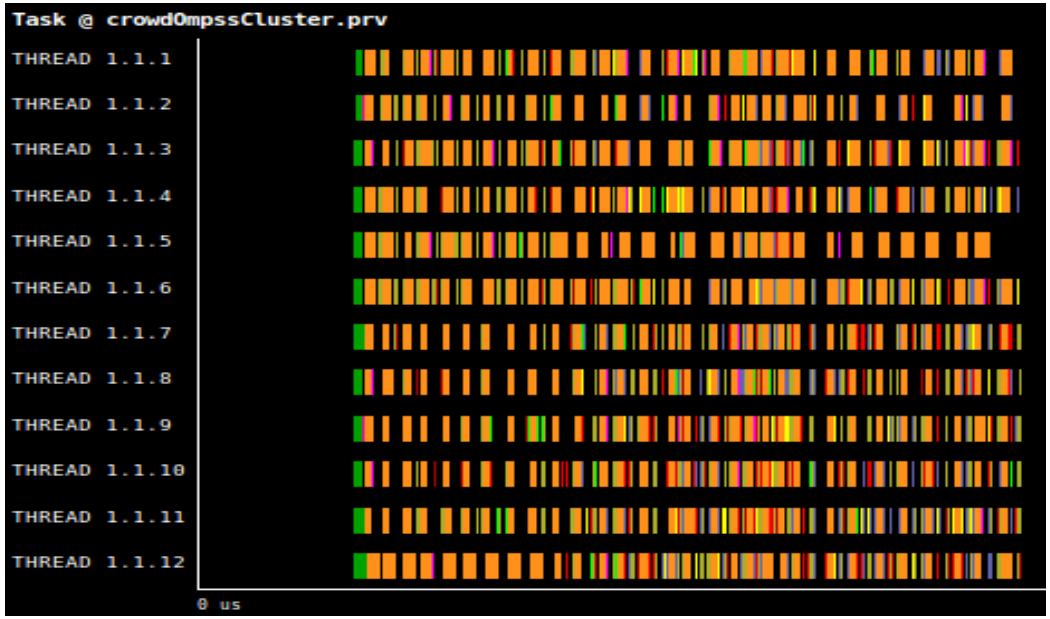
```
#pragma omp task inout(
[agents_total_buffer]agents[0;count_agents_total],
[agents_total_buffer]ids[0;count_agents_total],
[world_cells_block]world  )
void updatePositions(float4 *agents, float4 *ids, int *world, ...)     { ... }

bool runSimulation()
{ ... //Execution of task
     for (int i = 0; i < num_blocks; i++)
       updatePositions(agents[i], ids[i], world[i], ...);
...}
```

Call the function in a normal way

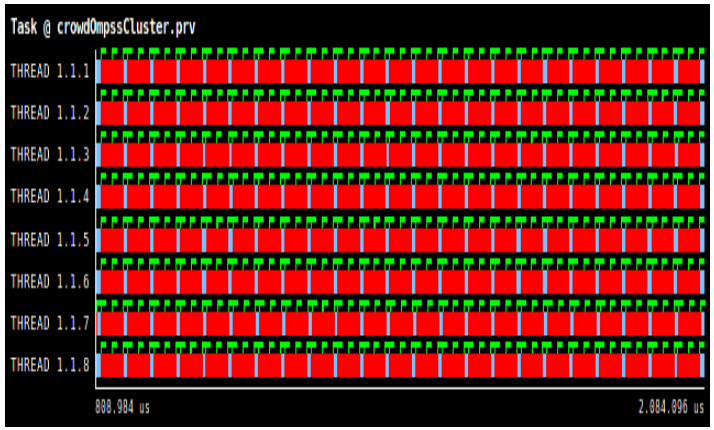
CROWD DEFINITION OF TASKS 2

Incremental definition of tasks
Flexible use of resources
Scaling through different number of CPU cores

Using 12 CPU cores
All functions converted to tasks



Using 8 CPU cores
Just 1 function converted to task



Using Extrae & Paraver
to get the traces

CROWD DEFINITION OF TASKS 3

Definition of GPU task

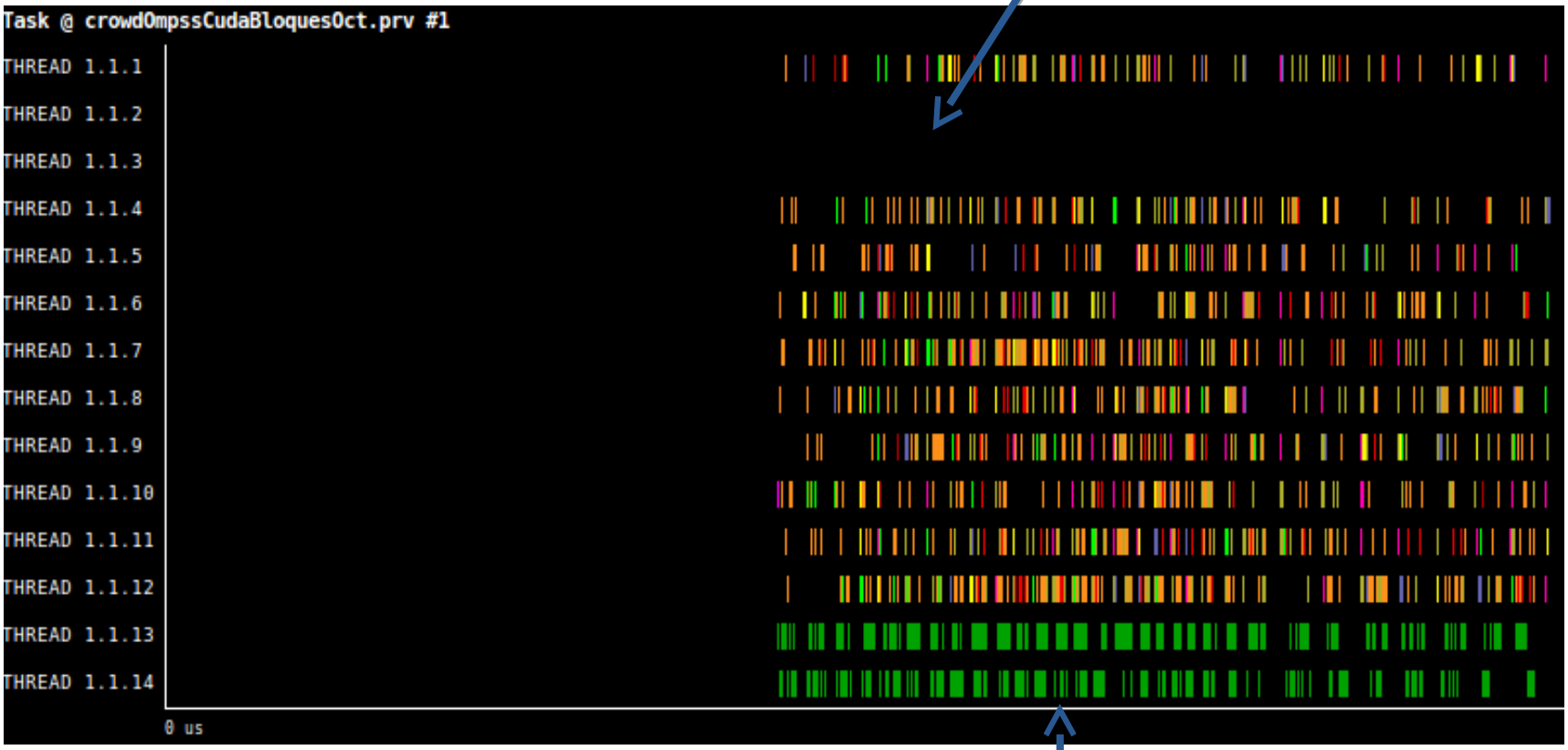
```
#pragma omp target device(cuda)
ndrange(2, (int)sqrt(count_agents_total), (int)sqrt(count_agents_total), 8, 8)
implements(updatePositions)
copy_deps
#pragma omp task inout(
    ([agents_total_buffer]agents)[0;count_agents_total],
    ([agents_total_buffer]ids)[0:count_agents_total],
    [world_cells_block]world )  label(refreshDataGPU)
extern "C" __global__ void updatePositionsGPU(float4 *agents, float4 *ids, int
*world, ...);

bool runSimulation()
{... //Task execution
   for (int i = 0; i < num_blocks; i++)
       updatePositions(agents[i], ids[i], world[i], ...);
 …}
```

Tasks must have the same parameters for both versions

CROWD DEFINITION OF TASKS 4

General view

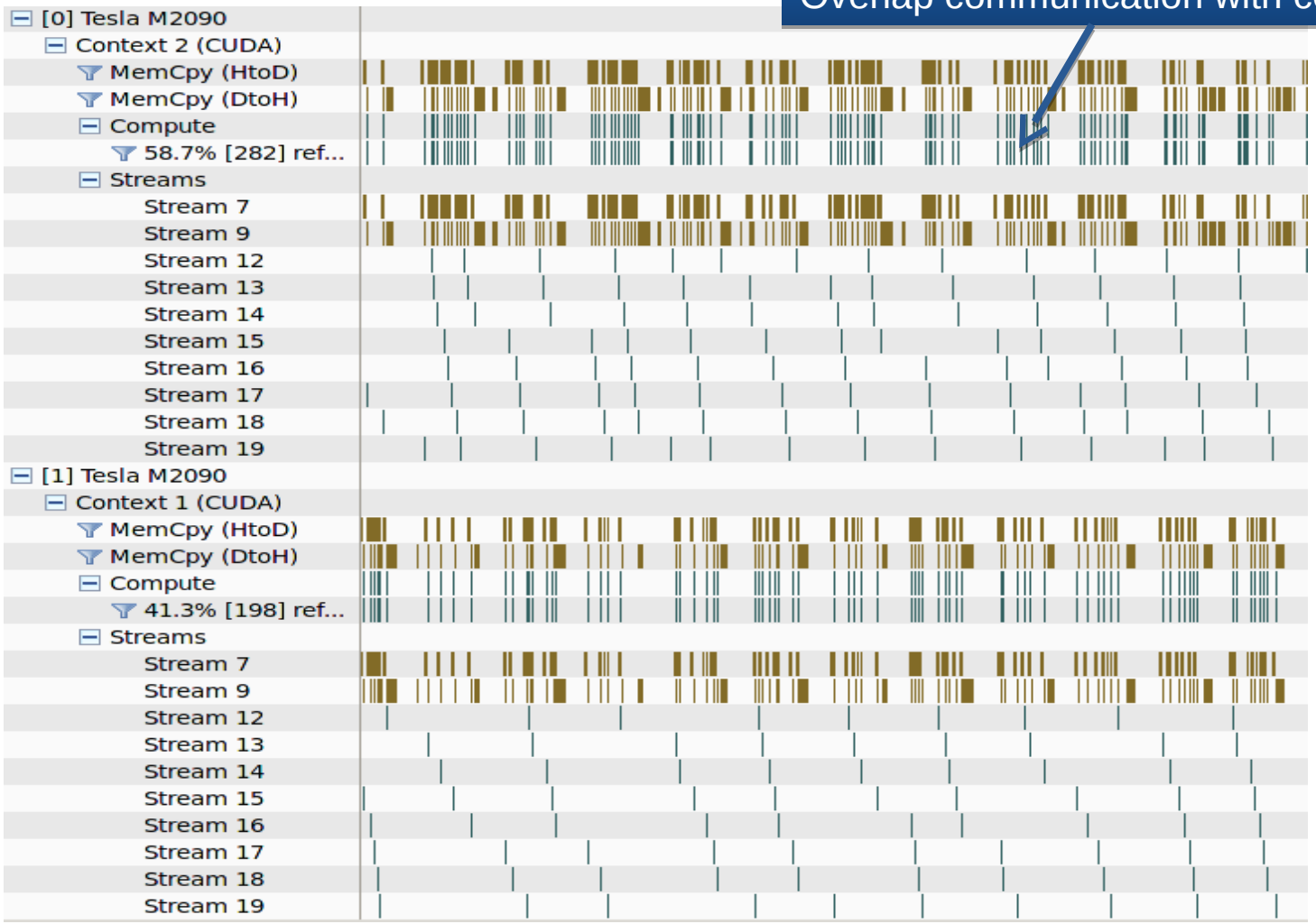
Two CPUs core to manage two GPUs



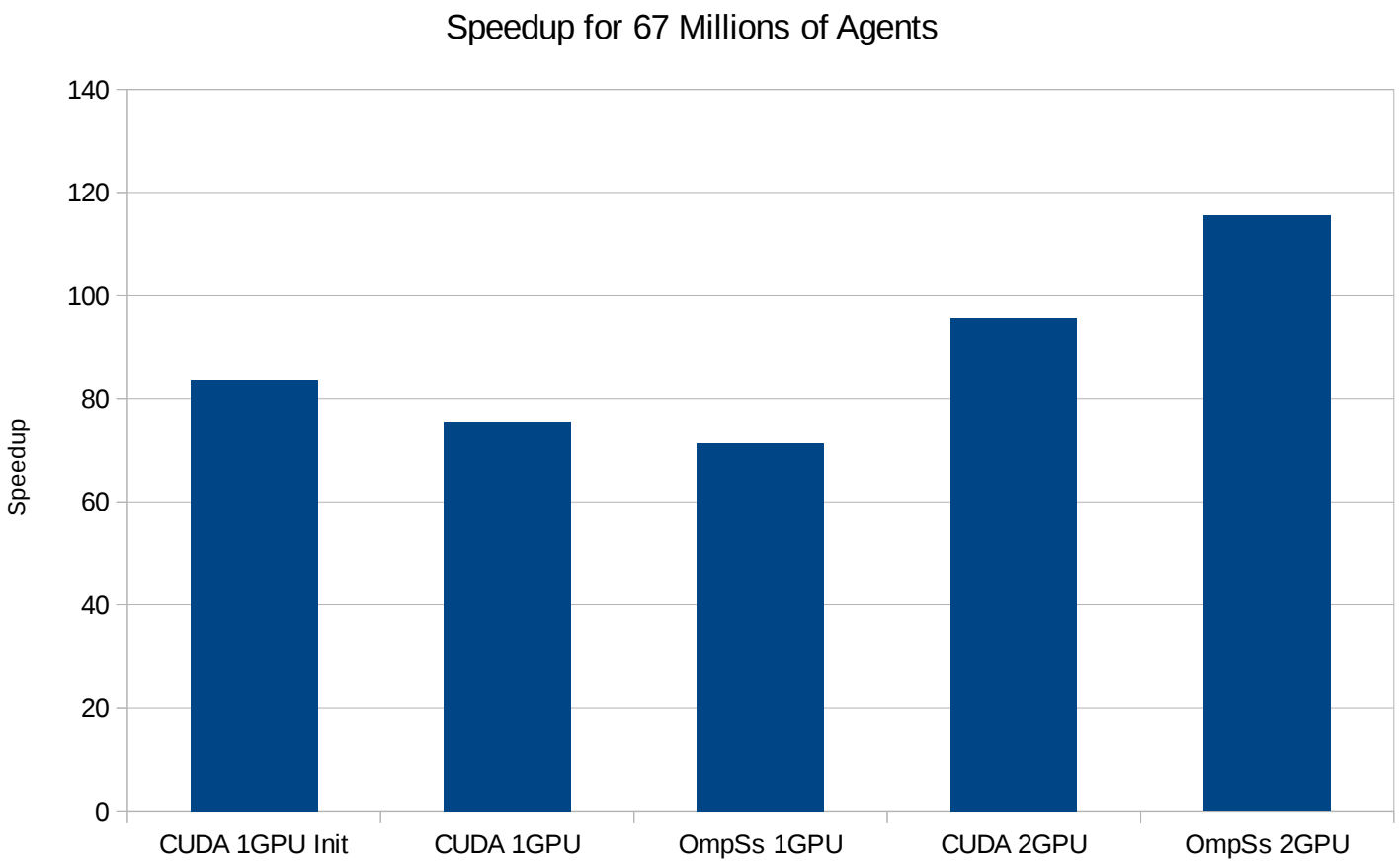
Execution of cuda kernels in the GPU

CROWD DEFINITION OF TASKS 5

Zoom to see the operations inside GPU



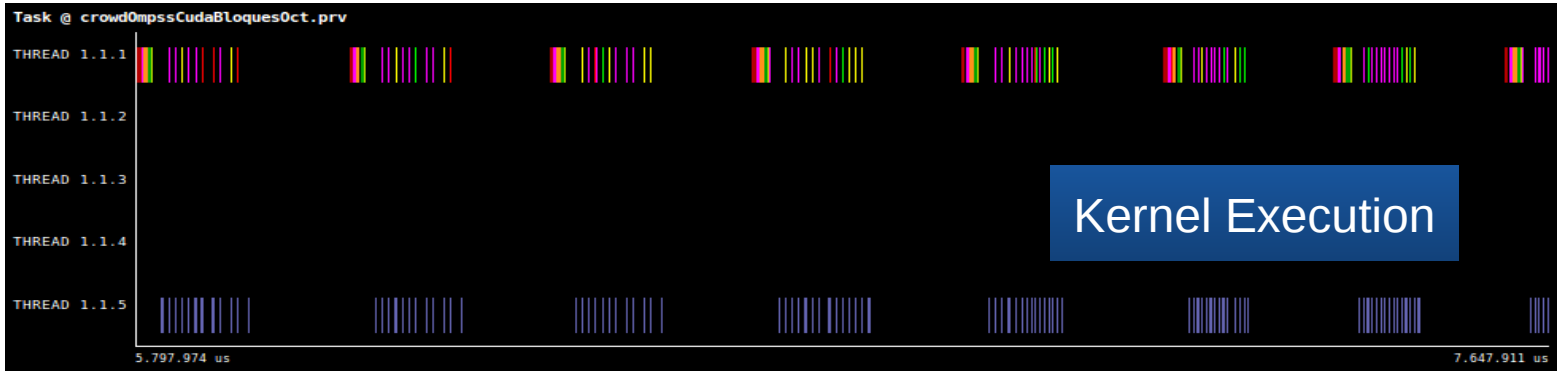
RESULTS

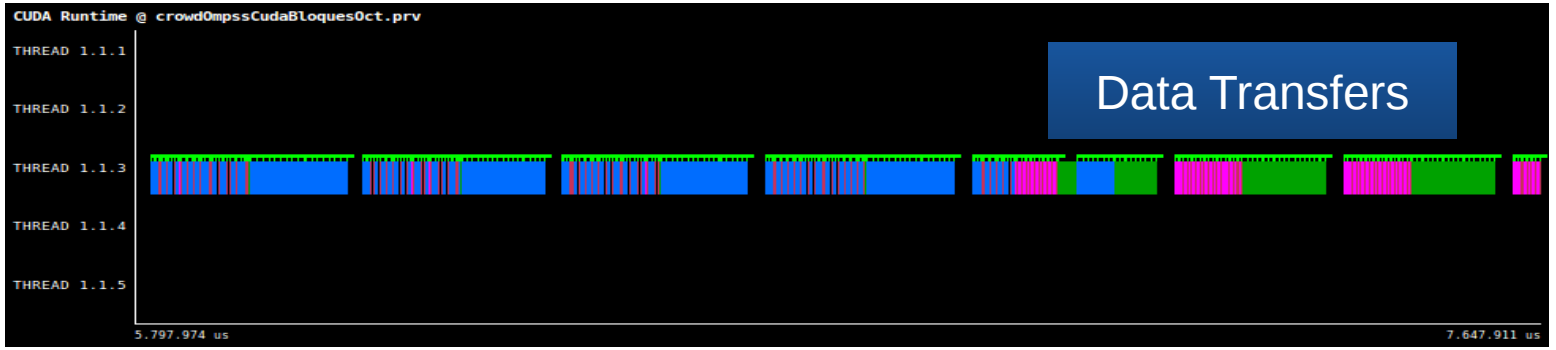


The speedup is calculated based on a core cpu execution of sequential version. Simulating 64 Millions of agents, the limit for the monolitic data version.

OMPSS AND OPENGL INTERACTION

OpenGL uses one GPU for rendering, while OmpSs uses the other one GPU for computation.
In OpenGL sharing GPUs between different threads is difficult.
Next generation OpenGL, called Vulkan will need the user to define tasks and schedule them. OmpSs will make this easier.





Flexible Interactive Parallel Architecture for Visualization

Can use different hardware setups by using combinations of:

- in situ visualization
- data streaming,
- virtualGL or web clients

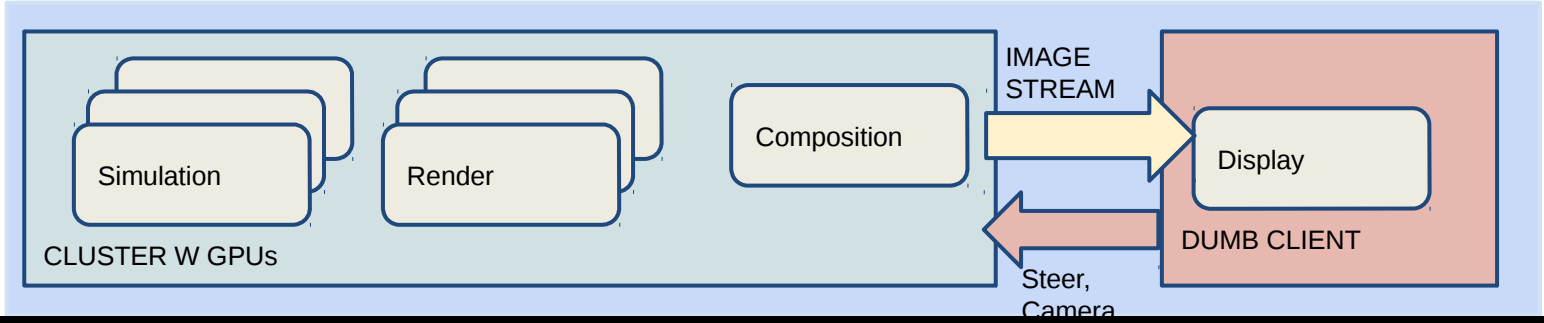
It has been used in different domains, and can thus exchange simulation and rendering engines as needed. For example:

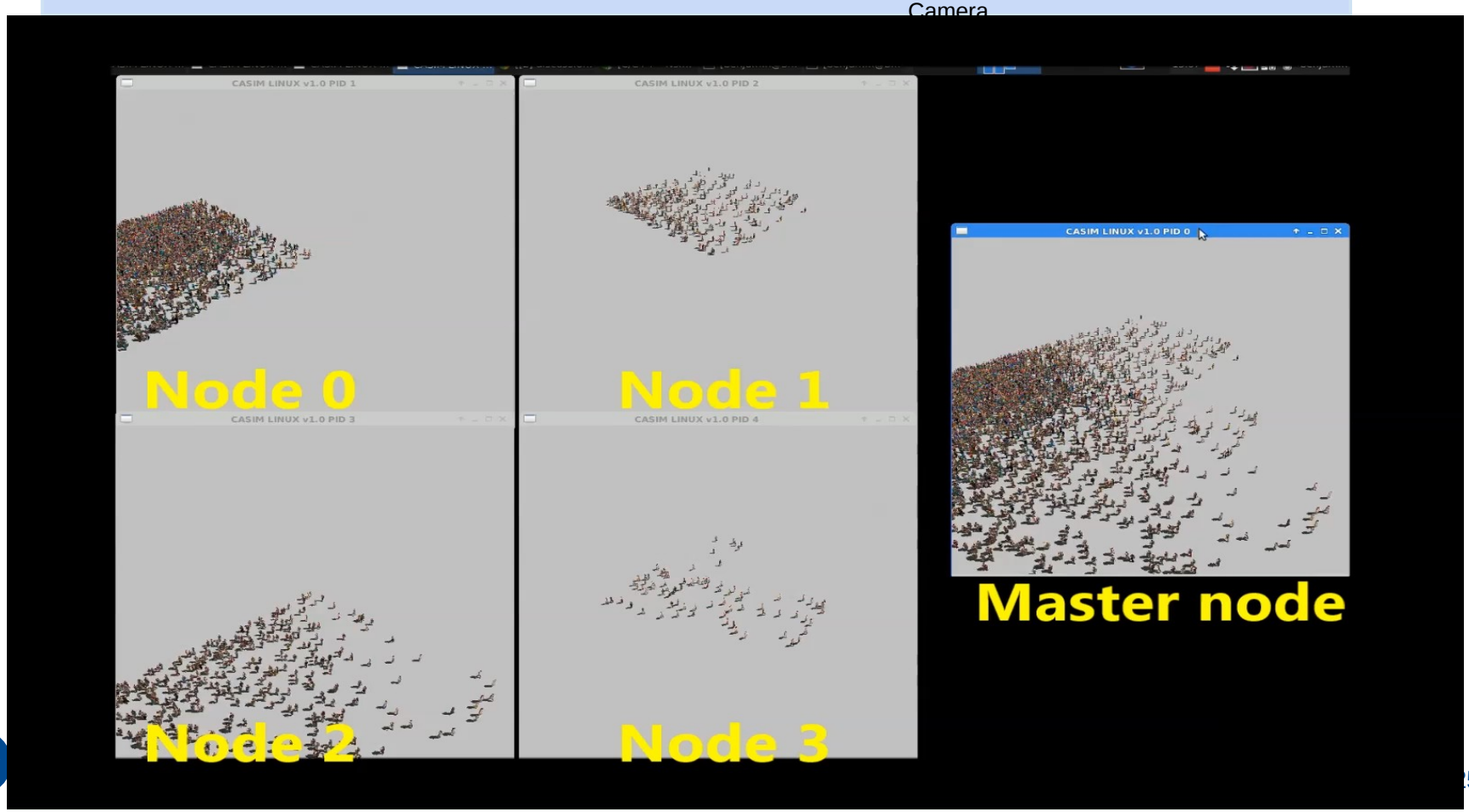
- Crowd simulation system developed with MPI+CUDA using varied and animated character crowd rendering engine using in-situ visualization in a GPU Cluster.
- Crowd simulation with Pandora: An HPC Agent-Based Modelling framework*, using crowd rendering engine via data streaming
- Fluid Visualization for Active Liquids. With UB Physics Dept. Better user interface, and support for state of the art visualization techniques, unavailable in VMD.
- Web accesible version of PELE - Protein Energy Landscape Exploration**
  Work in collaboration with group led by Fernando Cucchietti, and involving Computer Sciences, Life Sciences and CASE from BSC, as well as MOVING from UPC.

*http://www.bsc.es/computer-applications/pandora-hpc-agent-based-modelling-framework

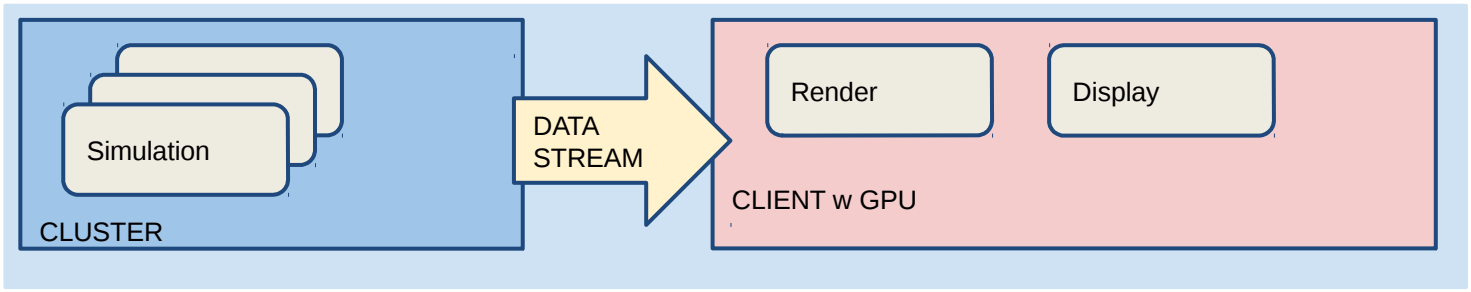
**https://pele.bsc.es/redmine/projects/pele-web-app

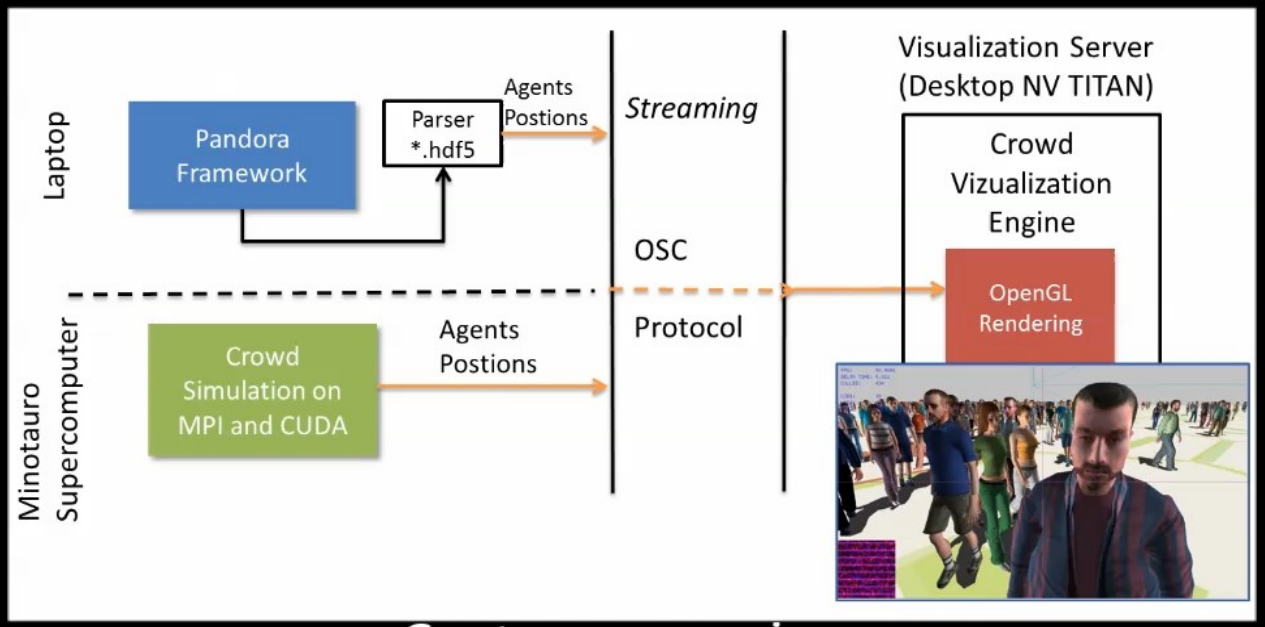
IN SITU


Simulation
Render
Composition
IMAGE STREAM
Display
DUMB CLIENT
CLUSTER W GPUs
Steer, Camera


Node 0
Node 1
Node 2
Node 3
Master node

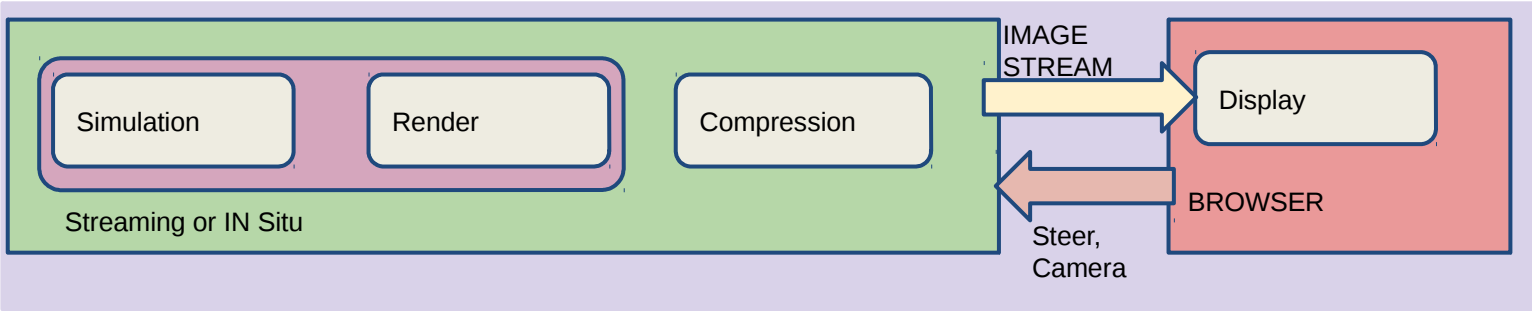
STREAMING

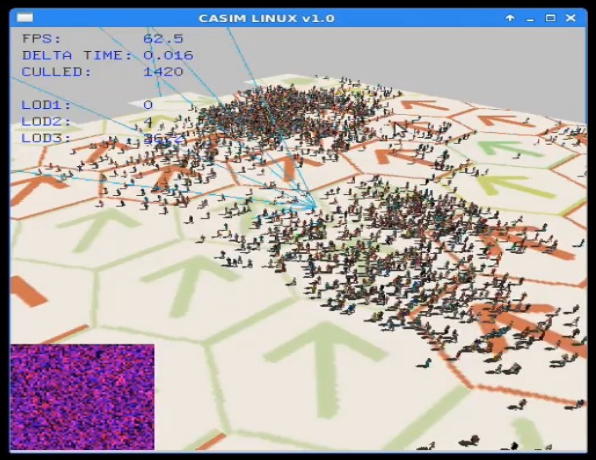


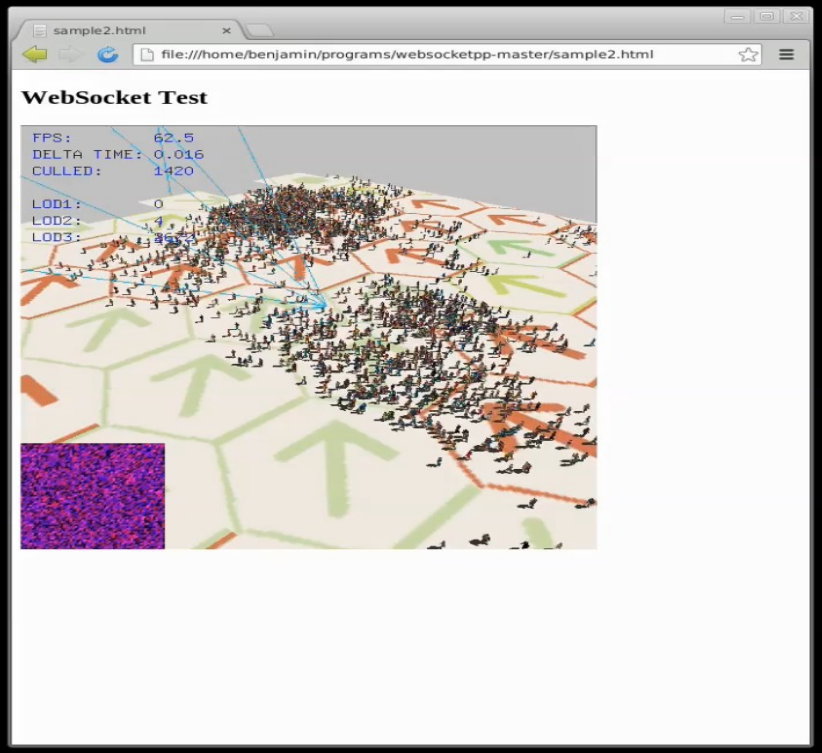


System overview

WEB


Simulation
Render
Compression
Streaming or IN Situ
IMAGE
STREAM
Display
BROWSER
Steer,
Camera


CASIM LINUX v1.0
FPS: 62.5
DELTA TIME: 0.016
CULLED: 1420
LOD1: 0
LOD2: 1


sample2.html
file:///home/benjamin/programs/websocketpp-master/sample2.html
WebSocket Test
FPS: 62.5
DELTA TIME: 0.016
CULLED: 1420
LOD1: 0
LOD2: 1

CONCLUSIONS

Algorithm with CUDA requires explicit operations:

- Device selection.
- Data transfers between host and devices.
- Flow control through queues and events.
- Among others.

  Difficult to program and prone to errors.

OmpSs:

- Facilitates use of all available GPUs in one node
- Allows us to make flexible use of resources, exploiting its full capacity
- Can scale the system to multiple CPUs or GPUs without modify the program.

For one GPU we get similar results than performing programming GPUs using CUDA more traditional way.
For two GPUs we get better results for OmpSs version.

OmpSs interacts with OpenGL for Visualization.

CONCLUSIONS 2

The visualization system has proven its flexibility in using

- different simulation and rendering engines
- the range of plattforms

FUTURE WORK

- We will work with OmpSs in a cluster comparing MPI+OmpSs+CUDA vs OmpSsCluster+CUDA
- We will study how Vulkan and OmpSs interact
- We will continue working on different schemes to couple simulation with visualization in the cluster that use Level of Detail rendering and partial composition.
- More use of compositing for coupling our crowd visualization system with:
  - other simulators (FlameGPU for example)
  - renderers and game engines (OptiX, Unreal)
  - GIS systems.

ACKNOWLEDGMENTS

- Nvidia and BSC through their CUDA Center of Excellence.
- Interministerial Commission of Science and Technology of Spain (CICYT).
- Oak Ridge Leadership Computing Facility ORNL
- Mexican Research and Council (CONACyT).
- And of course the development team of OmpSs.

OmpSs is open source! http://pm.bsc.es/ompss

For their support in this project.

MORE INFO

- Isaac Rudomin (BSC) isaac.rudomin@bsc.es
- Benjamin Hernandez (BSC,ORNL) hernandezarb@ornl.gov
- Hugo Perez (UPC-BSC) hugo.perez@bsc.es