Using OSPRay with Data-Distributed Applications

Will Usher, Jefferson Amstutz, Silvio Rizzi, Jim Jeffers and Valerio Pascucci
Data is either:

- Too big to fit on one node
- Already distributed (in situ)
- Both

Scientific visualization distributed rendering: Sort-last compositing

Each nodes’ region must be disjoint and convex to composite
Enable MPI-parallel applications to use OSPRay through “mpi_distributed” device

Ranks make independent API calls, specify local data

Committing the model synchronizes between ranks

Render frame works collectively to render the image
Using OSPRay’s Distributed API
OSPRay’s MPI Module

MPI parallel rendering functionality provided by the MPI module
Build with OSPRAY_MODULE_MPI=ON

Intel, Intel Core, Xeon, and Xeon Phi are trademarks of the Intel Corporation in the U.S. and other countries. Other product names and brands may be claimed as property of others.
OSPRay’s MPI Module

Provides two MPI parallel devices

- MPIOffload – Data replicated rendering, non-parallel applications
- MPIDistributed – Data parallel rendering, parallel applications

DistributedRaycastRenderer – SciVis style raycaster for distributed rendering

DistributedModel – World model for distributed data (more later)

DistributedFrameBuffer
Rendering Distributed (Opaque) Particles

Each rank has some different set of particle data, passes this to OSPRay to setup an OSPGeometry

Particles are opaque, so no regions required – DistributedFrameBuffer will z-compose properly

Follow along with the code on Github:
Particle Distribution Layout
Initialize MPI (optional)
Load MPI module, create device
Create local spheres
Add to model
Create renderer
Create framebuffer
Render!

int provided = 0;
MPI_Init_thread(&argc, &argv,
MPI_THREAD_MULTIPLE, &provided);

int world_size, rank;
MPI_Comm_size(MPI_COMM_WORLD, &world_size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (ospLoadModule("mpi") != OSP_NO_ERROR) {
    throw std::runtime_error("Failed to load OSPRay MPI module");
}

OSPDevice dev = ospNewDevice("mpi_distributed");
ospDeviceSet1i(dev, "masterRank", 0);
ospDeviceSetStatusFunc(dev,
    [](const char *msg){
        std::cout << msg << "\n";
    });
ospDeviceCommit(dev);
ospSetCurrentDevice(dev);
Distributed Opaque Particles (simple.cpp)

- Initialize MPI (optional)
- Load MPI module, create device
- Create local spheres
- Add to model
- Create renderer
- Create framebuffer
- Render!

OSPData sphere_data = ospNewData(...);
ospCommit(sphere_data);
OSPData color_data = ospNewData(...);
ospCommit(color_data);

OSPGeometry spheres = ospNewGeometry("spheres");
ospSetData(spheres, "spheres", sphere_data);
ospSetData(spheres, "color", color_data);
ospSet1f(spheres, "radius", 0.25f);
ospSet1i(spheres, "bytes_per_sphere", ...);
ospSet1i(spheres, "offset_colorID", ...);
ospCommit(spheres);
Distributed Opaque Particles (simple.cpp)

Initialize MPI (optional)
Load MPI module, create device
Create local spheres
Add to model
Create renderer
Create framebuffer
Render!

OSPMModel model = ospNewModel();
ospAddGeometry(model, spheres);
ospCommit(model);
Distributed Opaque Particles (simple.cpp)

Initialize MPI (optional)
Load MPI module, create device
Create local spheres
Add to model
Create renderer
Create framebuffer
Render!

OSPRenderer ren = ospNewRenderer("mpi_raycast");
ospSet1i(ren, "spp", 1);
ospSet1f(ren, "bgColor", 1.f);
ospSetObject(ren, "model", model);
ospSetObject(ren, "camera", camera);
ospCommit(ren);
Distributed Opaque Particles (simple.cpp)

Initialize MPI (optional)
Load MPI module, create device
Create local spheres
Add to model
Create renderer
Create framebuffer
Render!

OSPFrameBuffer fb = ospNewFrameBuffer(img_size,
OSP_FB_SRGBA,
OSP_FB_COLOR);
ospFrameBufferClear(fb, OSP_FB_COLOR);
Distributed Opaque Particles (simple.cpp)

Initialize MPI (optional)
Load MPI module, create device
Create local spheres
Add to model
Create renderer
Create framebuffer
Render!

```cpp
ospRenderFrame(fb, ren, OSP_FB_COLOR);

// Final rendered image available on rank 0
if (rank == 0) {
    const uint32_t *img =
        static_cast<const uint32_t*>(
            ospMapFrameBuffer(fb, OSP_FB_COLOR));
    write_ppm("simple.ppm",
              img_size.x, img_size.y, img);
    ospUnmapFrameBuffer(img, fb);
}
```
Example Image with 8 Ranks

mpirun –np 8 ./simple

Image saved to “simple_particles.ppm
Caveat: Only Works for Opaque Geometry

Opaque geometry just needs Z-compositing, which the DistributedFrameBuffer can do automatically.

However, compositing volume and transparent geometry requires us to know the bounds of the regions being rendered.
Distributed Regions for Compositing
Distributed Regions

DistributedModel in the module accepts a list of “regions”, bounding what data each rank owns

Very flexible specification of regions, OSPRay will use for sort-last compositing

Any local volumes or geometry can exist in these regions! OSPRay only needs the bounds

For correct compositing: Up to you to ensure regions are disjoint
Distributed Regions: Mixed Volumes & Particles

Application Rank

Application Rank

Application Rank

Application Rank
Distributed Volume Rendering (regions.cpp)

Initialize MPI (optional)
Load MPI module, create device
Create local volume brick
Generate spheres in region
Add to model, set region
Create renderer
Create framebuffer
Render!

```c
int provided = 0;
MPI_Init_thread(&argc, &argv,
    MPI_THREAD_MULTIPLE, &provided);

int world_size, rank;
MPI_Comm_size(MPI_COMM_WORLD, &world_size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```
if (ospLoadModule("mpi") != OSP_NO_ERROR) {
    throw std::runtime_error("Failed to load OSPRay MPI module");
}

OSPDevice dev = ospNewDevice("mpi_distributed");
ospDeviceSet1i(dev, "masterRank", 0);
ospDeviceSetStatusFunc(dev,
    [](const char *msg){
        std::cout << msg << "\n";
    });
ospDeviceCommit(dev);
ospSetCurrentDevice(dev);
Initialize MPI (optional)
Load MPI module, create device
Create local volume brick
Generate spheres in region
Add to model, set region
Create renderer
Create framebuffer
Render!

OSPVolume vol = ospNewVolume("block_bricked_volume");
ospSetString(vol, "voxelType", "uchar");
ospSetVec3i(vol, "dimensions",
    (osp::vec3i&)dims);
ospSetVec3f(vol, "gridOrigin",
    (osp::vec3f&)grid_origin);
ospSetObject(vol, "transferFunction", ...);

std::vector<uint8_t> data(dims.x*dims.y*dims.z,
    static_cast<uint8_t>(rank));
ospSetRegion(vol, data.data(),
    osp::vec3i{0, 0, 0},
    (osp::vec3i&)volume_dims);
ospCommit(volume);
Distributed Volume Rendering (regions.cpp)

- Initialize MPI (optional)
- Load MPI module, create device
- Create local volume brick
- Generate spheres in region
- Add to model, set region
- Create renderer
- Create framebuffer
- Render!

```cpp
uniform_real_distribution<float> pos_x(
    bounds.lower.x + radius,
    bounds.upper.x - radius);

for (size_t i = 0; i < 50; ++i) {
    atoms.push_back(Particle(pos_x(rng),
                             pos_y(rng), pos_z(rng)));
}

OSPData sphere_data = ospNewData(...);

OSPGeometry spheres = ospNewGeometry("spheres");
ospSetData(spheres, "spheres", sphere_data);
ospSet1f(spheres, "radius", radius);
// Other parameters...
ospCommit(spheres);
```
Distributed Volume Rendering (regions.cpp)

- Initialize MPI (optional)
- Load MPI module, create device
- Create local volume brick
- Generate spheres in region
- Add to model, set region
- Create renderer
- Create framebuffer
- Render!

```c
OSPModel model = ospNewModel();
ospAddVolume(model, vol);
ospAddGeometry(model, spheres);

box3f bounds(grid_origin, grid_origin + vec3f(dims));
OSPData region = ospNewData(2, OSP_FLOAT3, &bounds);
ospSetData(model, "regions", region);
ospCommit(model);
```
Distributed Volume Rendering (regions.cpp)

Initialize MPI (optional)
Load MPI module, create device
Create local volume brick
Generate spheres in region
Add to model, set region
Create renderer
Create framebuffer
Render!

OSPRenderer ren = ospNewRenderer("mpi_raycast");
ospSet1i(ren, "spp", 1);
ospSet1f(ren, "bgColor", 1.f);
ospSetObject(ren, "model", model);
ospSetObject(ren, "camera", camera);
ospCommit(ren);
Distributed Volume Rendering (regions.cpp)

Initialize MPI (optional)
Load MPI module, create device
Create local volume brick
Generate spheres in region
Add to model, set region
Create renderer
Create framebuffer
Render!

OSPFrameBuffer fb = ospNewFrameBuffer(img_size, OSP_FB_SRGBA, OSP_FB_COLOR);
ospFrameBufferClear(fb, OSP_FB_COLOR);
Distributed Volume Rendering (regions.cpp)

Initialize MPI (optional)
Load MPI module, create device
Create local volume brick
Generate spheres in region
Add to model, set region
Create renderer
Create framebuffer
Render!

ospRenderFrame(fb, ren, OSP_FB_COLOR);

// Final rendered image available on rank 0
if (rank == 0) {
    const uint32_t *img =
        static_cast<const uint32_t*>(
            ospMapFrameBuffer(fb, OSP_FB_COLOR));

    write_ppm("simple.ppm",
        img_size.x, img_size.y, img);
    ospUnmapFrameBuffer(img, fb);
}
Example Image with 8 Ranks

```bash
mpirun -np 8 ./regions
```

Image saved to “regions_sample.ppm”
Acknowledgements

This work was supported in part by NSF: CGV: Award:1314896, NSF CISE ACI-0904631, DOE/ Codesign P01180734, DOE/SciDAC DESC0007446, CCMSC DE-NA0002375, and PIPER: ER26142 DE-SC0010498. Additional support comes from the Intel Parallel Computing Centers program. This material is also based upon work supported by the Department of Energy, National Nuclear Security Administration, under Award Number DE-NA0002375.

The Argonne Leadership Computing Facility is a DOE Office of Science User Facility supported under contract DE-AC02-06CH11357.