A High-productivity Framework for Multi-GPU Computing of Weather Prediction Code ASUCA

Takashi Shimokawabe *(Tokyo Institute of Technology)*

shimokawabe@sim.gsic.titech.ac.jp
http://www.sim.gsic.titech.ac.jp/Japanese/Member/shimokawabe

The weather prediction code demands large computational performance to achieve fast and high-resolution simulations. Skillful programming techniques are required for obtaining good parallel efficiency on GPU supercomputers. Our framework-based weather prediction code ASUCA has achieved good scalability with hiding complicated implementation and optimizations required for distributed GPUs, contributing to increasing the maintainability; ASUCA is a next-generation high-resolution meso-scale atmospheric model being developed by the Japan Meteorological Agency. Our framework automatically translates user-written stencil functions that update grid points and generates both GPU and CPU codes. User-written codes are parallelized by MPI with intra-node GPU peer-to-peer direct access. These codes can easily utilize optimizations such as overlapping technique to hide communication overhead by computation.

1 Introduction

Numerical weather prediction is one of the major applications in high-performance computing and is accelerated on GPU supercomputers. Obtaining high-performance using thousands of GPUs often needs skillful programming. The Japan Meteorological Agency is developing a next-generation high-resolution meso-scale weather prediction code ASUCA. We are implementing it on a multi-GPU platform by using a high-productivity framework. This poster presents our proposed framework and its performance evaluation.

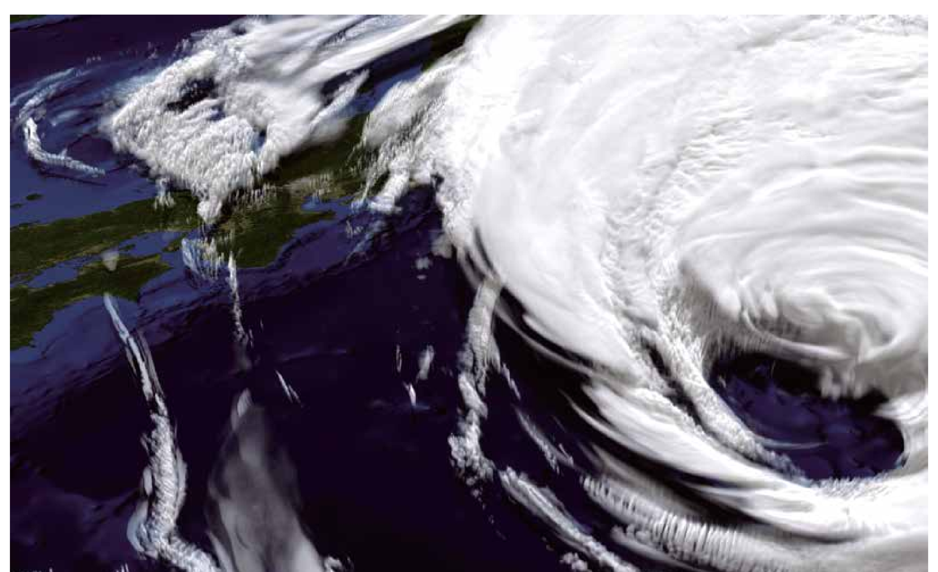


Figure 1: ASUCA simulation

2 Overview of Framework

- The proposed framework is designed for stencil applications with explicit time integration running on regular structured grids.
- The framework is intended to execute user programs on NVIDIA's GPUs and CPU.
- The framework is written in the C++ language and CUDA and can be used in the user code developed in the C++ language.
  --> Improving portability of both framework and user code and cooperation with the existing codes.
- The framework allows us to write multi-GPU code without considering handling multiple GPUs on a single process.
- To perform stencil computations on grids, the programmer only defines C++ functions that update a grid point, which is applied to entire grids by the framework.

Reference

[1] T. Shimokawabe, T. Aoki and N. Onodera **"High-productivity Framework on GPU-rich Supercomputers for Operational Weather Prediction Code ASUCA,"** *in Proceedings of the 2014 ACM/IEEE conference on Supercomputing (SC'14)*, New Orleans, LA, USA, Nov 2014.

3 Framework Implementation

Structure of Framework

- The framework supports multiple GPU computing.
- Optimized parallelization:
  Inter-node parallelization: MPI library
  Intra-node parallelization: OpenMP

This framework parallelizes not parts of GPU computation in the user code but the entire user code from beginning to end including memory allocation and time integration loop.

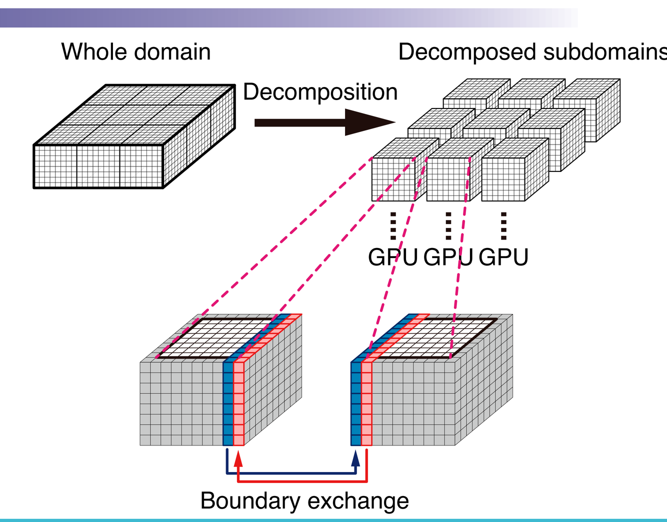


Figure 2: Multi-GPU computing of mesh-based computation.

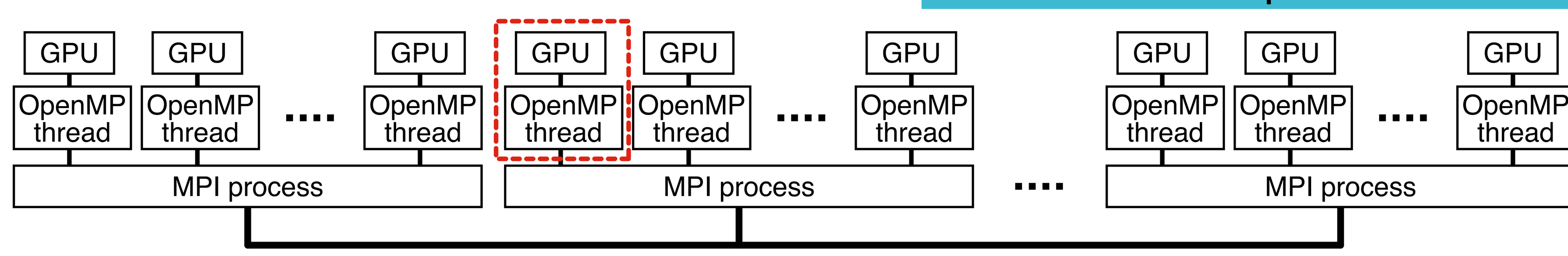


Figure 3: Multi-GPU computing by using both MPI and OpenMP.

Stencil Computation on Grids

In order to execute stencil computation on grids, the programmer must describe functions that update a grid point. The framework provides C++ classes that apply user-written functions to grids. The user-written functions are executed on grids sequentially for CPU and in parallel for GPU using CUDA's global kernel functions.

GPU-GPU communication

Intra-node GPU-GPU communication

- Multi-GPU calculations within a same node are performed by an MPI process with several OpenMP threads, each of which is assigned to a single GPU.
- This communication is performed by just a copy between the memories of two different GPUs using cudaMemcpy.
- GPUDirect peer-to-peer access is used when it is supported by two GPUs.

Inter-node GPU-GPU communication

- Three steps of boundary data exchange from GPU to GPU:
  (1) Data transfer from GPU to CPU using the CUDA runtime library
  (2) Data exchange between nodes with the MPI library
  (3) Data transfer back from CPU to GPU with the CUDA runtime library

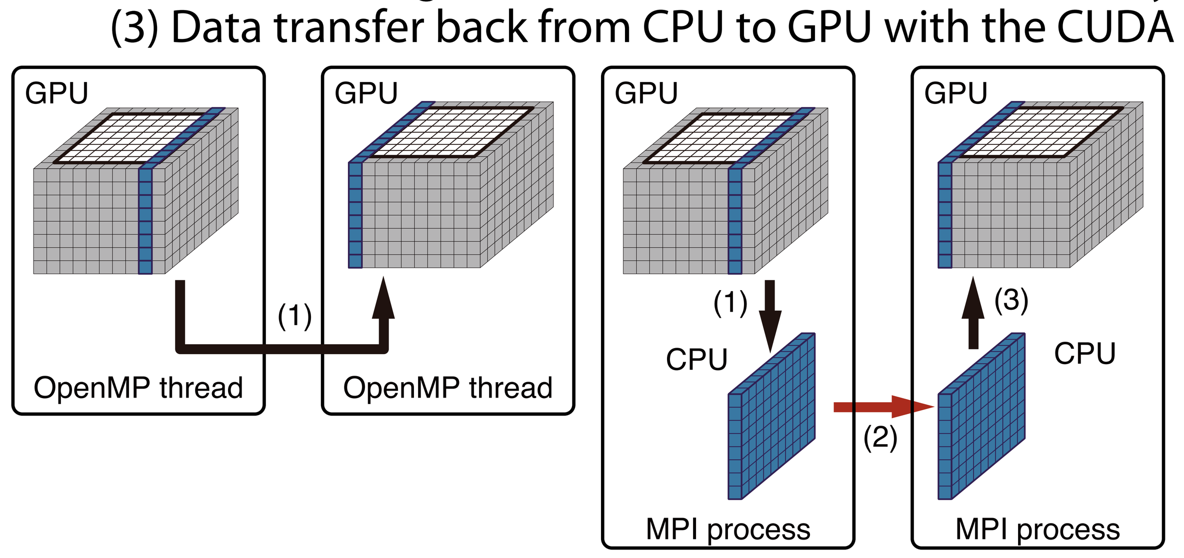


Figure 4:
left: Intra-node GPU-GPU communication by the OpenMP threads.
right: Inter-node GPU-GPU communication by MPI through host memory.

4 Programming Model

Parallelizing User Code

- User-written main code is executed in all OpenMP threads created by an OpenMP parallel directive.

```
DomainGroup domain_group(rank, &manager);
domain_group.run(main_run);
```

User-written main code , including memory allocation and time integration loop, runs in OpenMP parallel

Stencil Computation

- User-written function (C++ funtor) that updates a grid point
- ArrayIndex3D represents the coordinate of the point where this function is applied.

Diffusion computation

```
struct Diffusion3d {
  __host__ __device__
  void operator()(const ArrayIndex3D &idx,
    float ce, float cw, float cn, float cs, float ct, float cb, float cc,
    const float *f, float *fn) {
    fn[idx.ix()] = cc*f[idx.ix()] + ce*f[idx.ix<1,0,0>()] + cw*f[idx.ix<-1,0,0>()]
                                  + cn*f[idx.ix<0,1,0>()] + cs*f[idx.ix<0,-1,0>()]
                                  + ct*f[idx.ix<0,0,1>()] + cb*f[idx.ix<0,0,-1>()];
}};
```

The functor is executed over all grid points by Loop3D provided by the framework.

```
Loop3D loop3d(nx+2*mgnx, mgnx, mgnx, ny+2*mgny, mgny, mgny, nz+2*mgnz, mgnz, mgnz);
loop3d.run(Diffusion3d(), ce, cw, cn, cs, ct, cb, cc, f, fn);
```

User-written function — Parametres are provided to the user-written function.

GPU-GPU communication

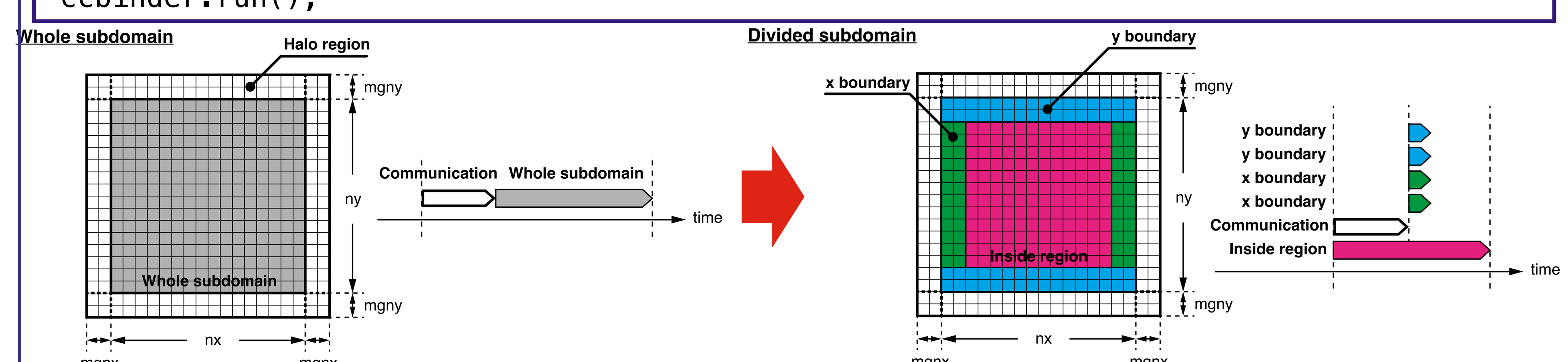
- Boundary regions of arrays appended by BoundaryExchange::append are exchanged when BoundaryExchange::transfer is executed.
- Boundary regions are automatically specified by BoundaryExchange class using Domain class, which has information of the computational domain.

```
BoundaryExchange *exchange = domain.exchange();
exchange->append(f);
exchange->transfer();
```

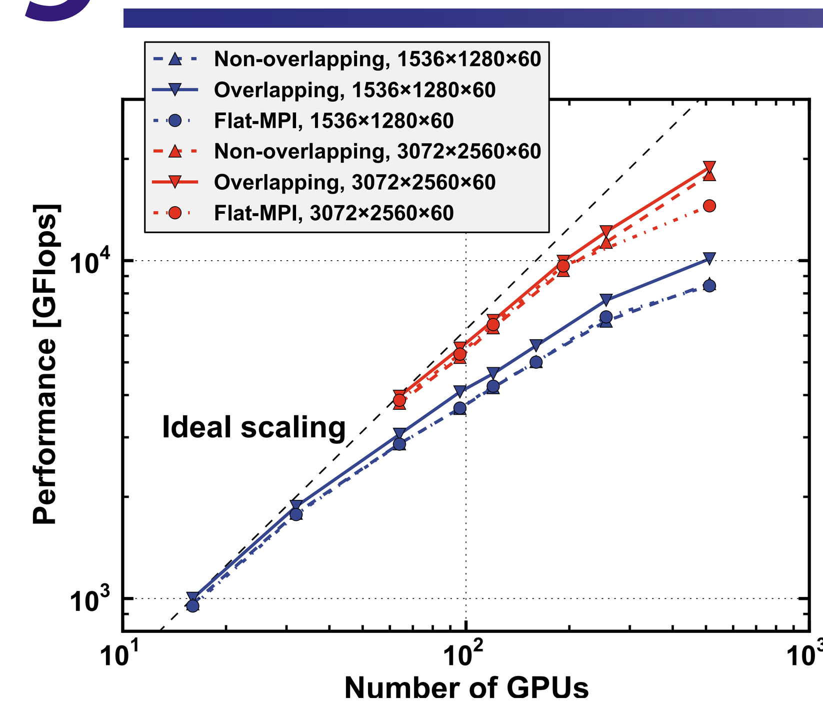
Overlapping method

- CompCommBinder class
  dividing five regions (in 2D)
  executes Stencil functions and communication in parallel

```
BoundaryExchange *exchange = domain.exchange();
exchange->append(f);
CompCommBinder<Loop3D> ccbinder(exchange);
ccbinder.set_post_func(&loop,
      create_funcholder<Loop3D>(Diffusion3d(),ce, cw, cn, cs ,ct, cb, cc, f, fn));
ccbinder.run();
```



5 Performance results



TSUBAME 2.5 supercomputer at the Tokyo Institute of Technology

- Total 4224 NVIDIA Tesla K20X GPUs
- Each node of TSUBAME 2.5
  - 3 Tesla K20X GPUs attached to the PCI Express bus 2.0 × 16 (8 GB/s)
  - 2 sockets of the Intel CPU Xeon X5670(Westmere-EP) 2.93 GHz 6-core
  - 2 QDR InfiniBand

Figure 5: Strong scaling results of the framework-based ASUCA on TSUBAME2.5. The overlapping method improves the overall performance.