National Institute for Environmental Studies, Japan

00:00:00

Introducing an Implementation of Parallel Computing for Lagrangian Modeling of Particle Dispersion in the Atmosphere

Jiye ZENG, Hideaki Nakajima, and Hitoshi MUKAI

00:00:00

Introduction

- In a typical application of Lagrangian modeling of particle dispersion in the atmosphere, the number of particles in one release is usually more than 10,000 and the time step for integrating differential equations ranges from a few to tens of seconds.

- Under these conditions, it takes a long time to get solutions when the required number of model run is large. In some situations, it is necessary to produce results quickly. Implementing parallelization not only is a way to meet the demand with limited computing resources in such occasions, but also makes it possible to do more experiments with a model within limited time.

- In this presentation, we introduce an implementation of parallel computing for Lagrangian Modeling using OpenMP and NVIDIA's Graphic Processor Unit (GPU), show the performance gains with a PC and a node in a supercomputer, and discuss the pro and cons of the GPU.

- Finally, we present a case study for the radiation leak from the Fukushima No.1 Nuclear Power Plant of Japan, which was caused by the earthquake and tsunami on 11 March 2011.

Parallelization

- A model may adopt one or both of the two parallelisms: data parallelism and task parallelism.
- Data parallelism is also known as loop-level parallelism. It focuses on distributing the data across parallel computing nodes and is achieved when each processor performs the same task on different pieces of distributed data.
- Task parallelism is also known as function parallelism and control parallelism. It focuses on distributing execution processes across different parallel computing nodes and is achieved when each processor performs a different task on the same or different data.
- Because the movement of particles in a Lagragian model is modulated by random processes, the use of input data in 3D space for any single particle is unpredictable; therefore, a special type of data parallelism, shared memory data parallelism, becomes our choice for parallelization. It is achieved by using different processors in a PC or a node of a supercomputer to perform the same task on different pieces of data.

OpenMP

OpenMP (Open Multi-Processing) is an application program interface for multi-threaded, shared memory parallelism. Taking the following code block as a simple example, an OpenMP-capable compiler will examine the preprocessor directive and generate a binary executable that activates multiple threads at run time to do the adding in parallel. The only requirement for OpenMP to work correctly is the loop includes no racing condition, e.g., C[i+1]=A[i]+B[i].

```
void VecAdd(float* A, float* B, float* C, int n)
{
#pragma omp parallel for

    for (int i=0; i<n; ++i) {
        C[i] = A[i] + B[i];
    }
}
```

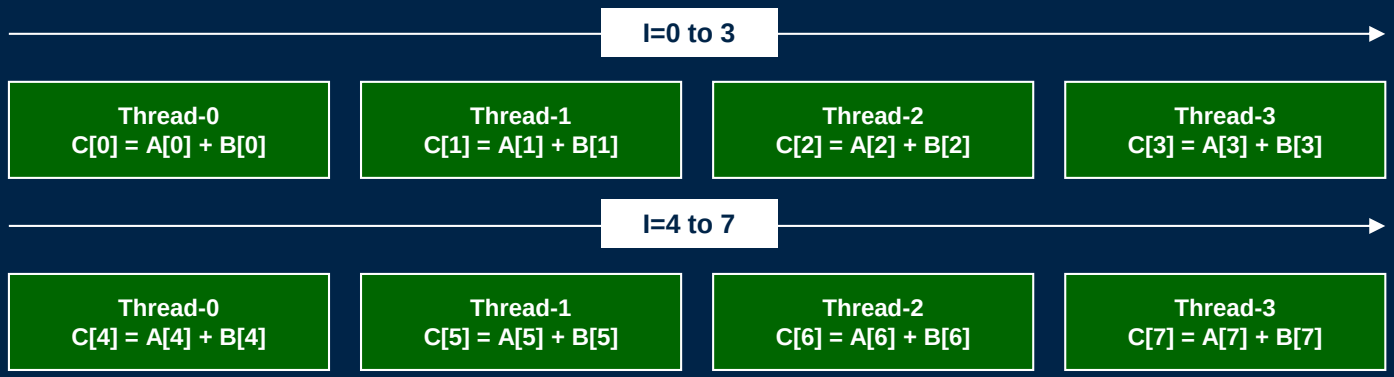


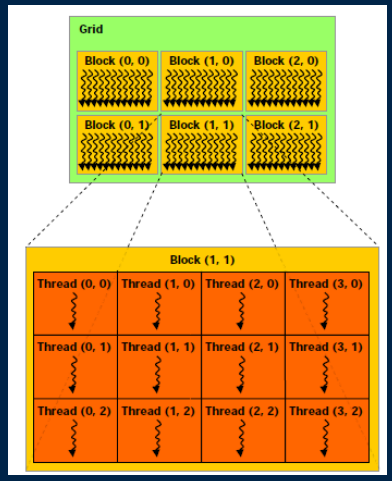
Illustration of executing the loop by four threads.

NVIDIA GPU

- The GPU is especially well-suited for data-parallel computations. NVIDIA provides a parallel computing architecture for the GPU, called CUDA. As an example of C for CUDA extension, the following code block illustrates how the host computer of a GPU device activates at least 256 threads in the GPU device to perform pair-wise additions of arrays with each thread doing one addition.

```
// Kernel function definition
__global__ void VecAdd(float* A, float* B, float* C, int n)
{
    int i = threadIdx.x;
    if (i<n) C[i] = A[i] + B[i];
}

int main()
{
    // Declare and Initialize A, B, C here.
    ……
    // Kernel invocation (C for CUDA extention)
    VecAdd<<<1, 256>>>(A, B, C, n);
}
```

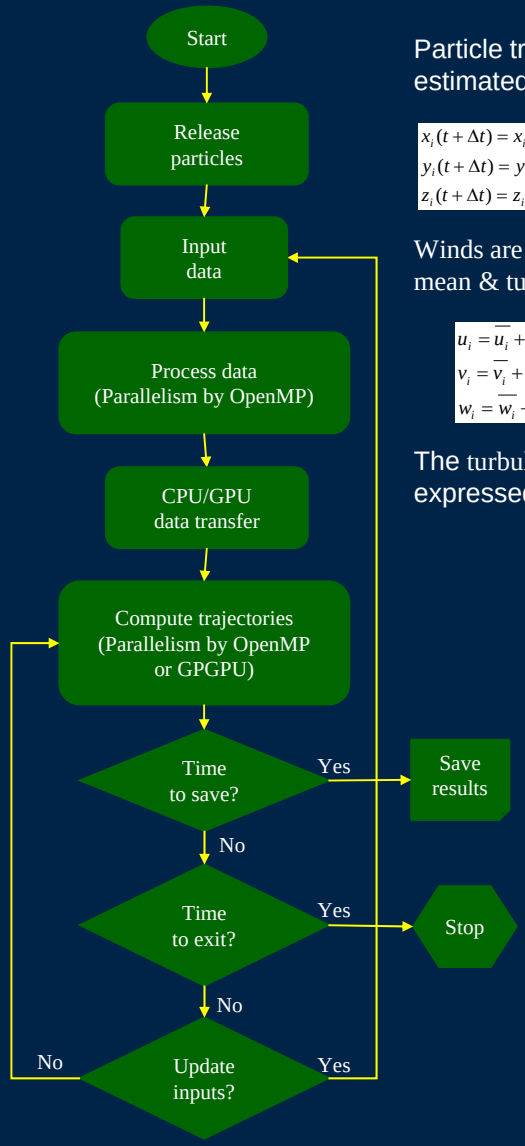


- Because the GPU can process a great number of threads efficiently, it is an ideal device for simulating particle dispersion: individual particles can be handled by different threads in pair-wise approach.

OpenMP & GPU

00:00:00

- **We use the combination of OpenMP and GPU in parallelization: parallel data processing in the host computer is achieved by OpenMP and parallel computing of particle dispersion is done in GPU.**
- **The kernel functions and data structure are designed in a way that the same source code can be used by the GPU and host computer processors.**
- **This approach simplifies debugging code: we can do debugging mainly in the host computer and then do verification in the.**



Particle trajectories (x,y,z) are estimated from winds (u,v,w):

$$x_i(t+\Delta t) = x_i(t) + u_i(t)\Delta t$$
$$y_i(t+\Delta t) = y_i(t) + v_i(t)\Delta t$$
$$z_i(t+\Delta t) = z_i(t) + w_i(t)\Delta t$$

Winds are decomposed into the mean & turbulent components:

$$u_i = \overline{u_i} + u_i'$$
$$v_i = \overline{v_i} + v_i'$$
$$w_i = \overline{w_i} + w_i'$$

The turbulent components are expressed as:

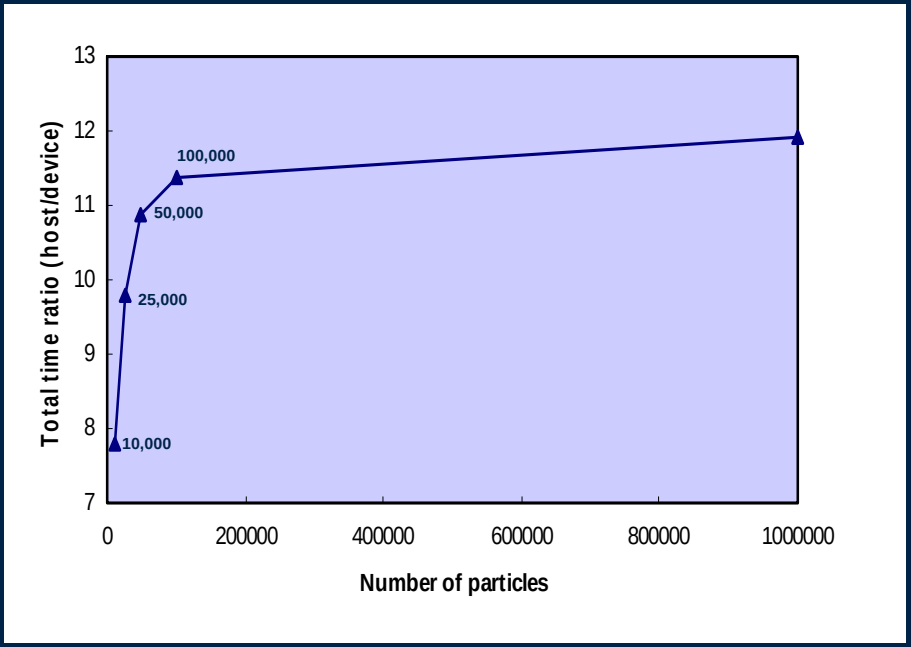
$$\left(\frac{u'}{\sigma_u}\right)_{k+1} = \left(1-\frac{\Delta t}{\tau_{Lu}}\right)\left(\frac{u'}{\sigma_u}\right)_k + \left(\frac{2\Delta t}{\tau_{Lu}}\right)^{1/2}\zeta$$
$$\left(\frac{v'}{\sigma_v}\right)_{k+1} = \left(1-\frac{\Delta t}{\tau_{Lv}}\right)\left(\frac{v'}{\sigma_v}\right)_k + \left(\frac{2\Delta t}{\tau_{Lv}}\right)^{1/2}\zeta$$
$$\left(\frac{w'}{\sigma_w}\right)_{k+1} = \left(1-\frac{\Delta t}{\tau_{Lw}}\right)\left(\frac{w'}{\sigma_w}\right)_k + \frac{\sigma_w}{\partial z}\Delta t + \frac{\sigma_w}{\rho}\frac{\partial \rho}{\partial z}\Delta t + \left(\frac{2\Delta t}{\tau_{Lw}}\right)^{1/2}\zeta$$

Where

σ: standard deviation of turbulent wind;
τ: autocorrelation timescale;
ρ: air density;
ζ: normally distributed random number with zero mean and unit standard deviation.

NIES

Performance Test-1



According to the CUDA programming guide, the maximum number of active threads per multiprocessor is 768. Tesla C1060 has 30 multiprocessors, so it can have up to 23,040 active threads.

Conditions:

- GPU device: Tesla C1060, processor clock 1.3GHz, memory clock 800 MHz, memory size 4GB.
- OS: Windows XP 64bit edition
- Host computer: Intel Xeon CPU, 2.00GHz, 4GB RAM.
- Execution: Read data every 3 hours, interpolate data every minutes, time step is 10 seconds, back trajectory length is 3 days, save results every hour.

Performance Test-2

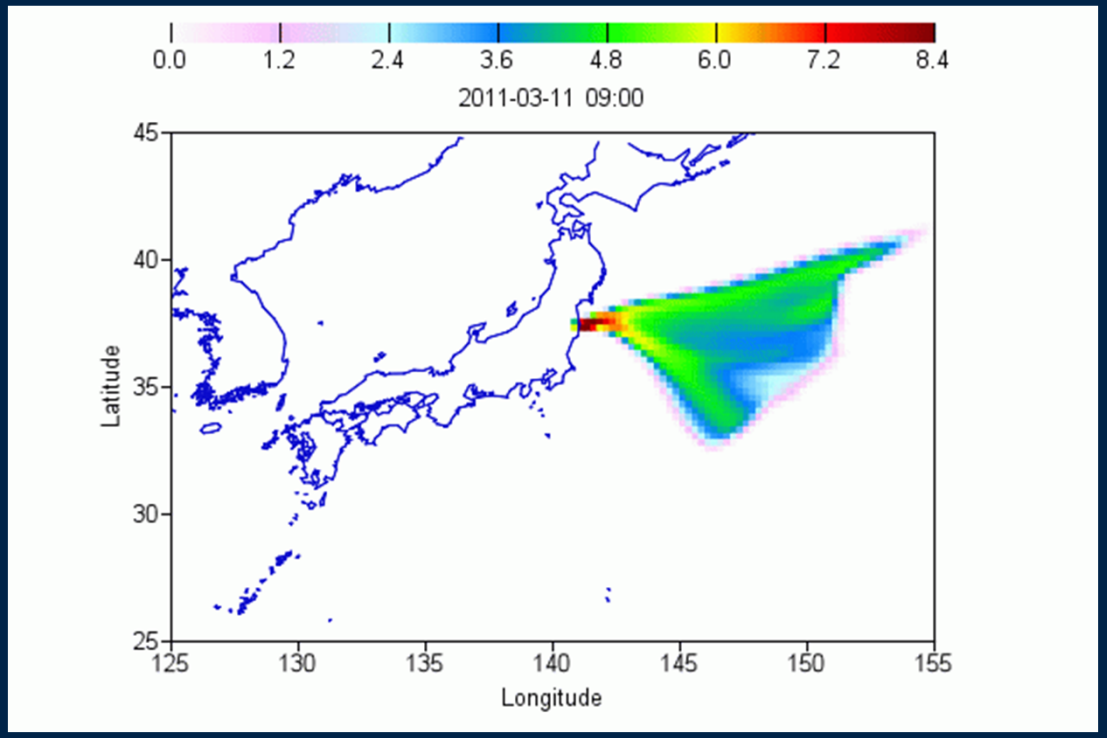
CPU Threads	GPU Use	Execution Time (s)	Time Ratio
1	No	604	1.0
8	No	87	6.9
1	Yes	28	21.6
8	Yes	15	40.3

Conditions:

- **GPU device: Tesla C2050, processor clock 1.15GHz, memory clock 1.5GHz, memory size 3GB.**
- **OS: CentOS 5.4**
- **Host computer: A node in the GOSAT RCF cluster. The system comprises 160 nodes, each having 2 quad-core CPUs and 2 NVIDIA Tesla C2050 GPUs.**
- **Execution: Read data every 3 hours, interpolate data every minutes, time step is 10 seconds, back trajectory length is 3 days, save results every hour.**

Case Study

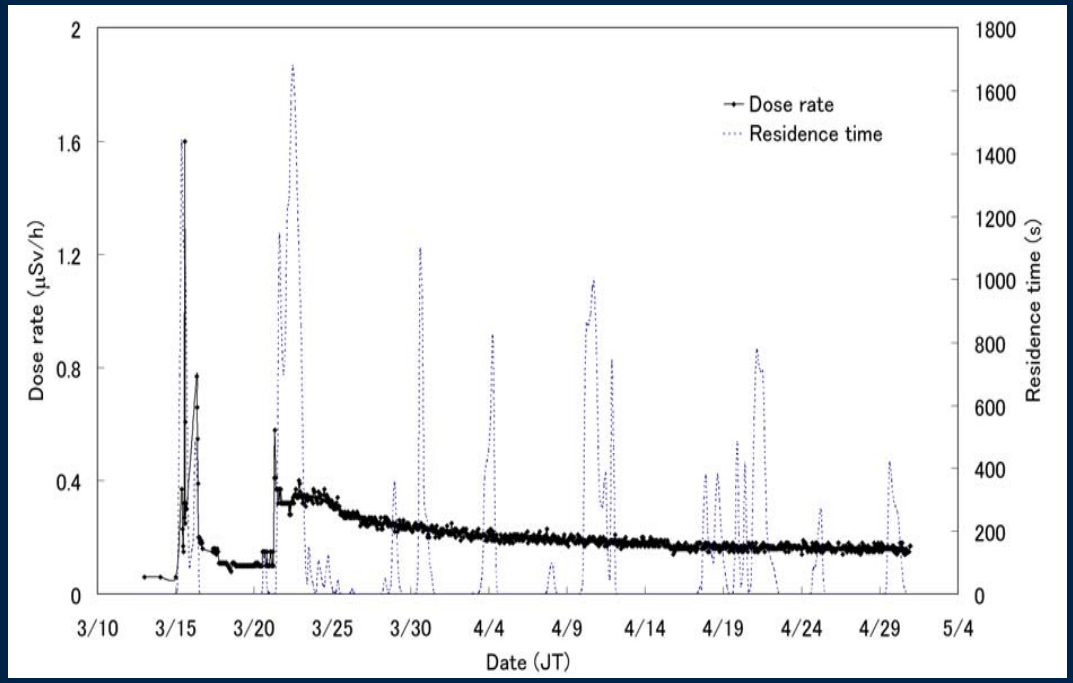
00:00:00



Color bar:
log(1+residence_time(s))
in 0.5x0.5 grids

- **We used the radiation leak from the Fukushima No.1 Nuclear Power Plant of Japan in a case study for testing the implementation. The lead was caused by the damage of the power plant due to the earthquake and tsunami on 11 March 2011.**
- **50,000 particles were released to the location of the power plant at time interval of 3 hours and their dispersion was simulated for 10 days.**

Case Study



- The figure shows the observed radiation dose rate and modeled residence time. The residence time is the mean of all 0.25x0.25 grids in a 1x1 grid area around the Tsukuba Center.
- The first two waves of particle arrival on March 15 and 21, indicated by long particle residence times, agree well with the observed surges of the dose rate. In other periods of long residence times after March 28, the dose rate did not show significant increases, indicating that after the explosions in the early days at the power plant, the radiation release to the atmosphere had been stable, even if not under control

Summary

- Our Lagrangian Model is based on FLEXPART. Results from the case study indicate that the model works well. The source code, a binary package for Windows, and utility programs for converting data and plotting results are available at

  http://db.cger.nies.go.jp/metex/flexcpp.html

- The parallelization has improved the performance up to 40 times for NVIDIA's GPU and even without the GPU, the performance gain is proportional to the number of processors in a PC or a node of a supercomputer.

- Parallelization for multi-core processors is relatively easy by using OpenMP, but Parallelization for the GPU requires major re-write of source code. Although some compilers can optimize existing source code automatically for the GPU, the performance gain is small, about two to three times, according to tests with a transport model by another group in our institute.

- The GPUs we used have limits on local memory. This leads to the failure of parallelizing the code for the most convection scheme. The C code comes from F77 to C conversion by using the F2C tool.

Suggestions

- Use or add a module to use a standard format for input and output. I recommend netCDF or HDF. My experiences with FLEXPART and GRIB: 1) Difficult to understand; 2) data providers may use different ID numbers for the same parameters; 3) incorrect meta information in encoded GRIB data.
- Make program usable for both pressure level and sigma level data.
- Make a common dataset available to the community so that beginners can get started easily and advanced users can compare different models easily. ECMWF data: not problem for EU, but problematic for low budget researchers. Open data: lack of parameters (NCEP reanalysis), limited period (GDAS1), incomplete (GFS), and the issue of parameter definition (sign of sensible heat flux).