Big Data Visual Analytics (CS 661)

Instructor: Soumya Dutta

Department of Computer Science and Engineering

Indian Institute of Technology Kanpur (IITK)

email: soumyad@cse.iitk.ac.in

Acknowledgements

- Some of the following slides are adapted from the excellent course materials and tutorials made available by:
  - Prof. Han-Wei Shen (The Ohio State University)

Study Materials for Lecture 8

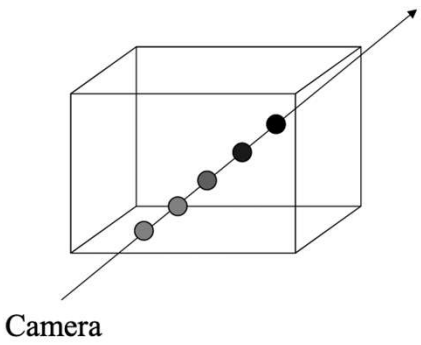
- Parallel Algorithms:
  - https://hpc.llnl.gov/documentation/tutorials/introduction-parallel-computing-tutorial
  - Parallel Volume Rendering Using Binary Swap Image Composition, Ma et al.
  - Parallel Volume Rendering on the IBM Blue Gene/P, Peterka et al.

Accelerating Volume Rendering

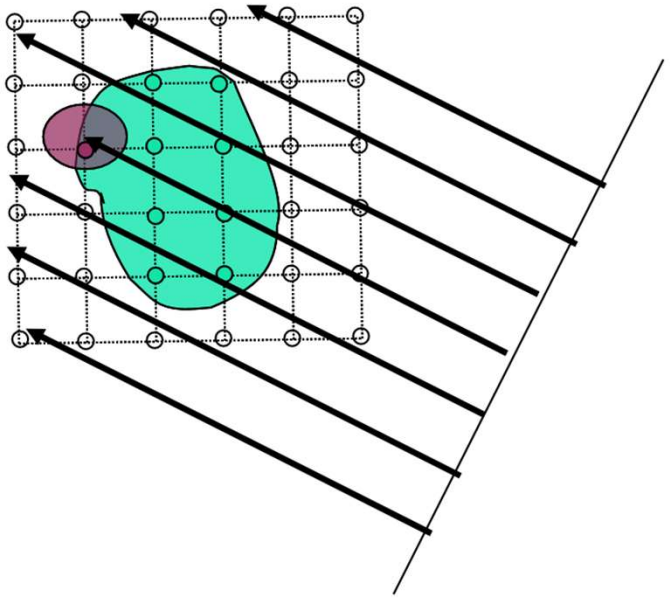
- Early ray termination
- Empty space skipping
- Adaptive sampling

Early Ray Termination

- In front to back compositing, we keep track of accumulated opacities separately
- We can stop ray traversal when accumulated alpha/opacity for a ray reaches 1 and nothing behind will be visible

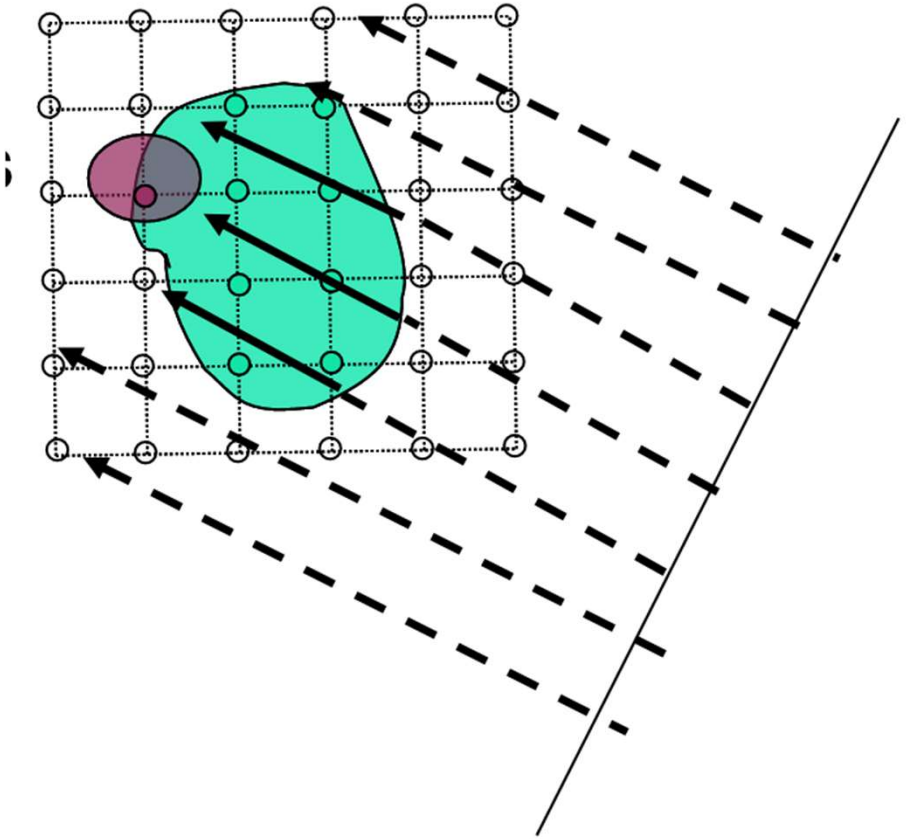


Current Alpha	Accumulated Alpha
0.2	0.2
0.4	0.4+(1- 0.4)*0.2=0.52
0.5	0.5+(1- 0.5)*0.52=0.76
0.4	0.4+(1- 0.4)*0.76=0.86
0.6	0.6+(1- 0.6)*0.86=0.94



Empty Region Skipping

- Skip Empty Cells
- Homogeneity acceleration
  - Approximate homogeneous regions with fewer sample points



Adaptive Sampling

- Increase sample density in high-gradient regions
- Decrease sample density in low-gradient/homogeneous regions

Dealing with Large-Scale Data via Parallel Data Processing and Visualization

Why Parallel Programming?

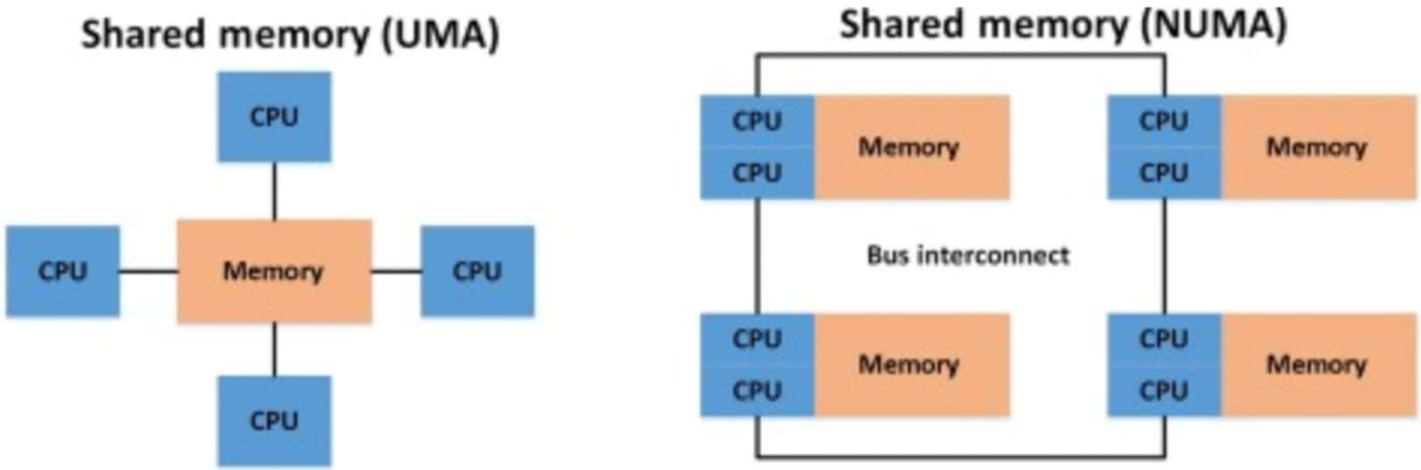
- Utilize the aggregated memory from all computing elements to store very large data sets
- Utilize the aggregated compute power to share the expensive task workload
- Utilize the aggregated I/O bandwidth, if a parallel file system is available
- Data computed from large scale simulations are typically stored in a file system connected to a supercomputer

Parallel Computing Paradigms

- Shared-memory parallelism
- Distributed-memory parallelism
- Hybrid parallelism

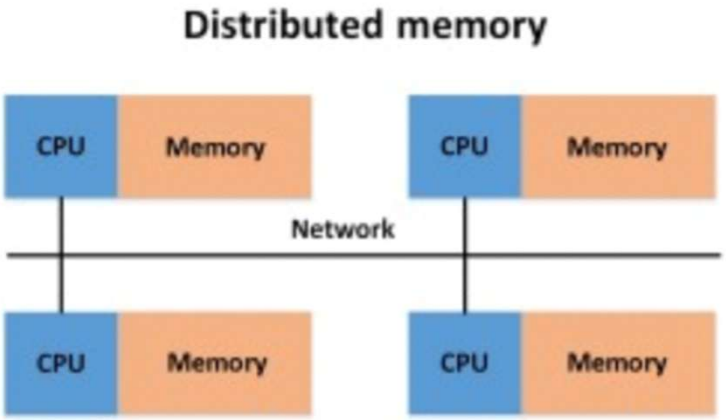
Shared Memory Parallelism

- Shared-memory parallelism
  - All PEs can access the shared memory, hence have the same view of the data
  - No explicit message passing is needed
  - Computation are done through multiple threads, or libraries such as OpenMP



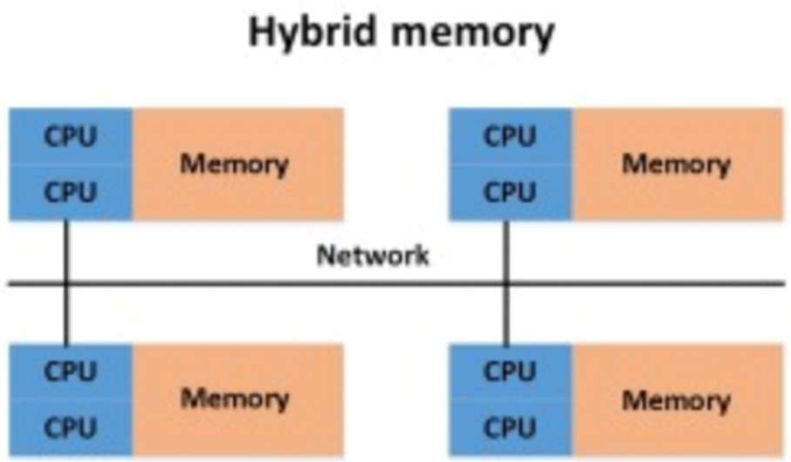
Distributed Memory Parallelism

- Distributed-memory parallelism
  - Data are distributed across the local memory of different Processing Element (PE)
  - Coordination among PEs is done through message passing using libraries such as MPI
  - Tasks are run on each core within each processing node



Hybrid Parallelism

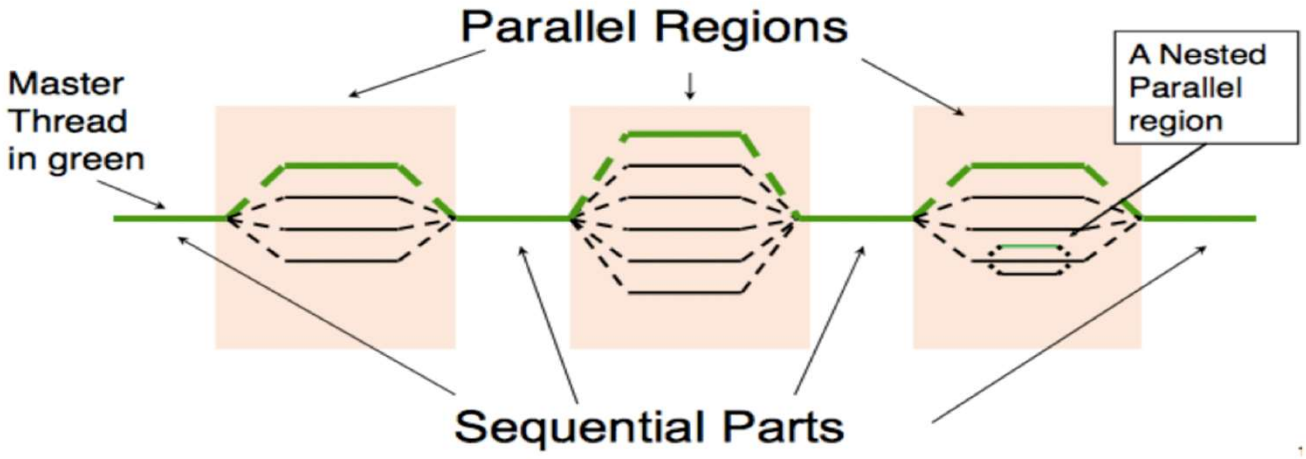
- Hybrid parallelism
  - Hybrid parallelism refers to a blend of distributed- and shared-memory parallel programming techniques within a single application
  - Each node has multiple threads running shared memory parallelism, but nodes have distributed memory and communicate with one another using message passing
  - Most modern clusters and supercomputers provide hybrid parallelism capability



Parallel Programming API – Shared Memory

- Open Multi-Processing (OpenMP) - Shared Memory Parallelism
- Latest OpenMP versions also supports GPUs

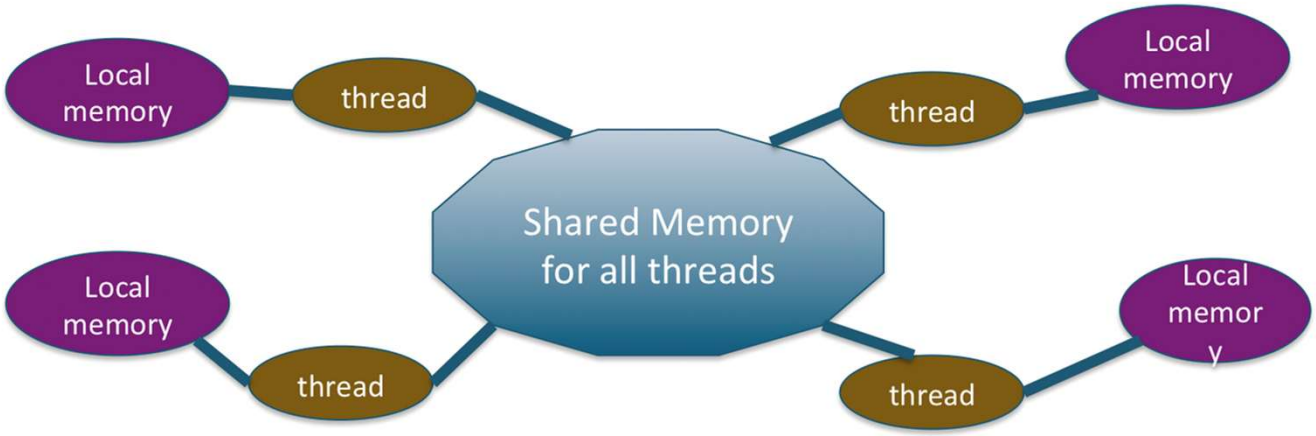
  https://www.openmp.org/



Parallel Programming API – Shared Memory

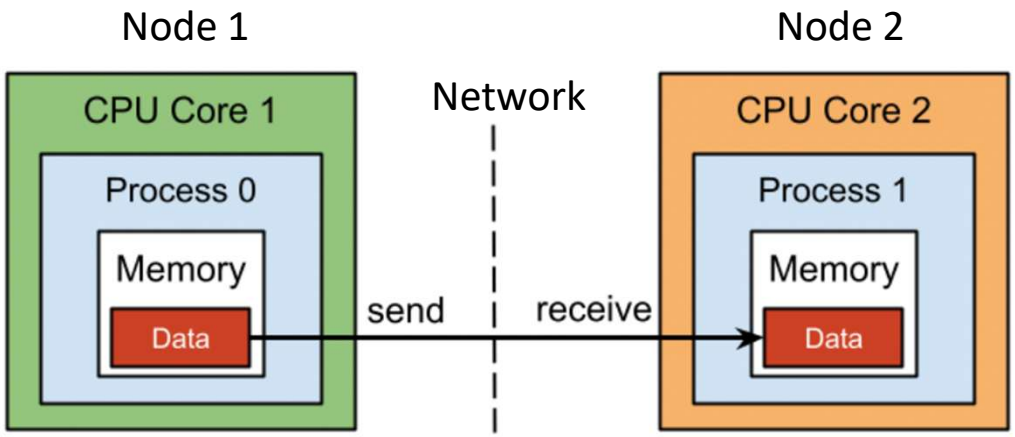
- Open Multi-Processing (OpenMP) - Shared Memory Parallelism
- Latest OpenMP versions also supports GPUs

  https://www.openmp.org/



Parallel Programming API – Distributed Memory

- Message Passing Interface (MPI) – Distributed Parallelism



Param Sanganak at IITK

Total number of nodes: 332 (20 + 312)

o Service nodes: 20 (Master+ Login+ Service+ Management)
o CPU only nodes: 150 (Total 7200 cores, 28800GB memory)
o GPU nodes: 20
o High Memory nodes:78

- Different implementations of MPI:
  - Open MPI (https://www.open-mpi.org/)
  - MVAPICH (https://mvapich.cse.ohio-state.edu/)
  - MPICH (https://www.mpich.org/)

OpenMP vs MPI

MPI	OpenMP
Available from different vendors and gets compiled on Windows, macOS, and Linux operating systems.	An add-on in a compiler such as a GNU compiler and Intel compiler.
Supports parallel computation for distributed-memory and shared-memory systems.	Supports parallel computation for shared-memory systems only.
A process-based parallelism.	A thread-based parallelism.
With MPI, each process has its own memory space and executes independently from the other processes.	With OpenMP, threads share the same resources and access shared memory.
Processes exchange data by passing messages to each other.	There is no notion of message-passing. Threads access shared memory.
Process creation overhead occurs one time.	It depends on the implementation. More overhead can occur when creating threads to join a task.

OpenMP vs MPI: Hello World

```c
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[])
{
    int nthreads, tid;

    #pragma omp parallel private(nthreads, tid)
    {
        tid = omp_get_thread_num();
        nthreads = omp_get_num_threads();
        printf("Hello World Thread %d / %d\n", tid, nthreads);


    }

    return 0;
}
```

```
Hello World Thread 1 / 2
Hello World Thread 2 / 2
```

```c
#include <mpi.h>
#include <stdio.h>

int main(int argc, char** argv) {
    // Initialize the MPI environment
    MPI_Init(NULL, NULL);

    // Get the number of processes
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    // Get the rank of the process
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // Get the name of the processor
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(processor_name, &name_len);

    // Print off a hello world message
    printf("Hello world from processor %s, rank %d out of %d processors\n",
           processor_name, world_rank, world_size);

    // Finalize the MPI environment.
    MPI_Finalize();
}
```

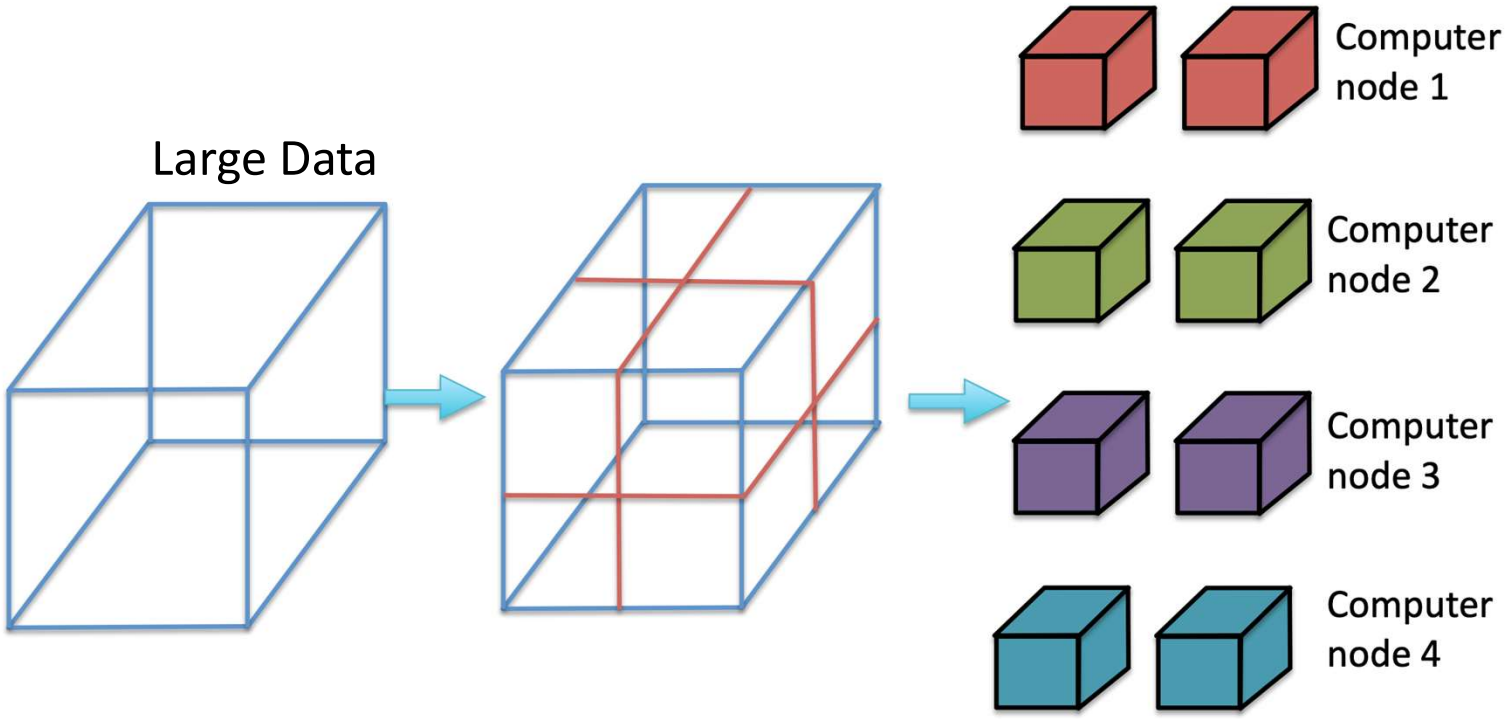
```
Hello world from processor ea72304b6e5c, rank 0 out of 4 processors
Hello world from processor ea72304b6e5c, rank 1 out of 4 processors
Hello world from processor ea72304b6e5c, rank 2 out of 4 processors
Hello world from processor ea72304b6e5c, rank 3 out of 4 processors
```

Steps for a Parallel Distributed Algorithm

- Data Decomposition
- Distribute decomposed data to processors
- Parallelly process data and apply the algorithm
- Aggregate partial results from different processors to produce the final result

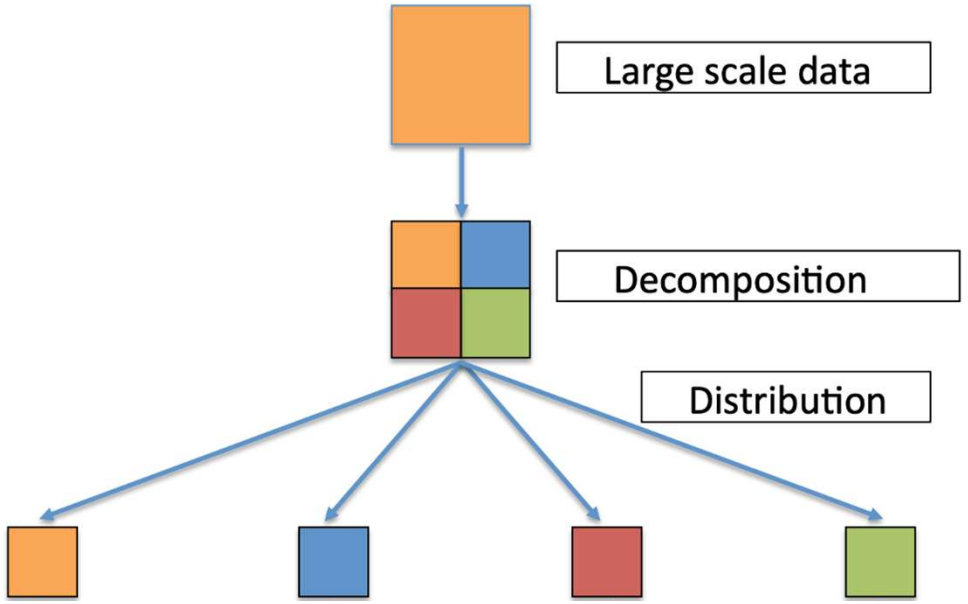
Data Decomposition

After decomposition, data blocks are distributed to processors



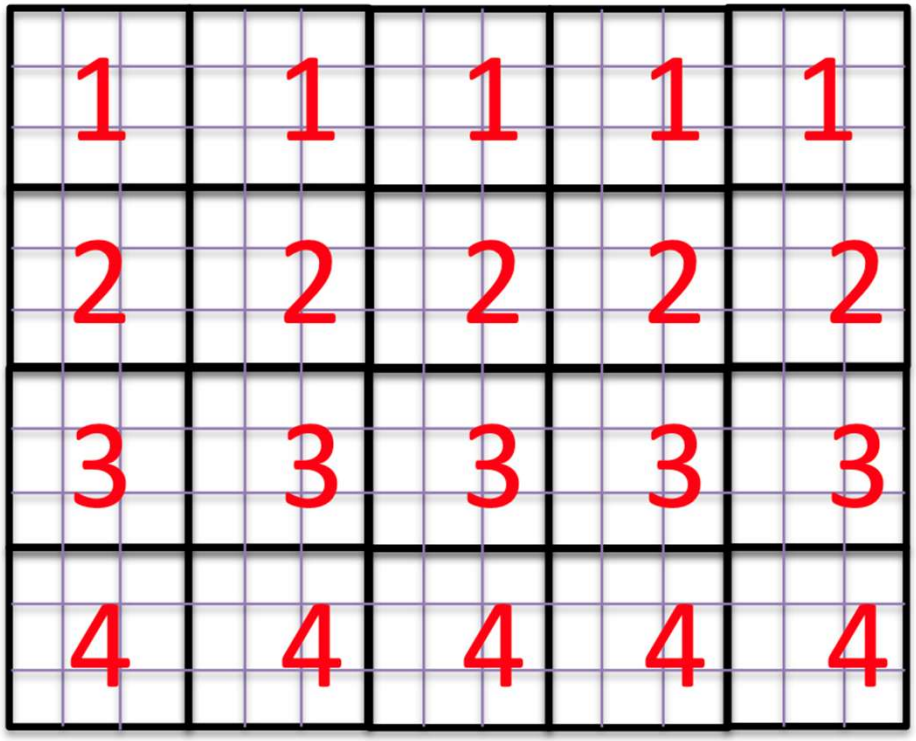
Data Distribution

- After decomposition, data blocks need to be distributed to processors
- Proper distribution is critical for the overall performance and minimizing overhead
- What overhead?
  - Load imbalance
  - Communication of data over network among processors
    - Synchronization



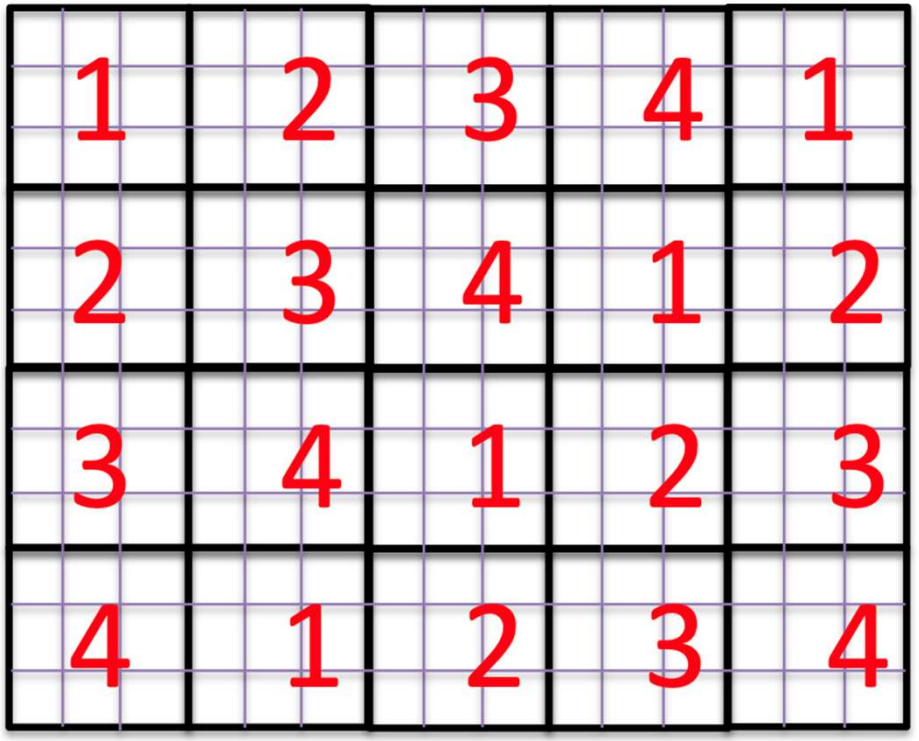
Data Distribution Techniques

- Contiguous distributions



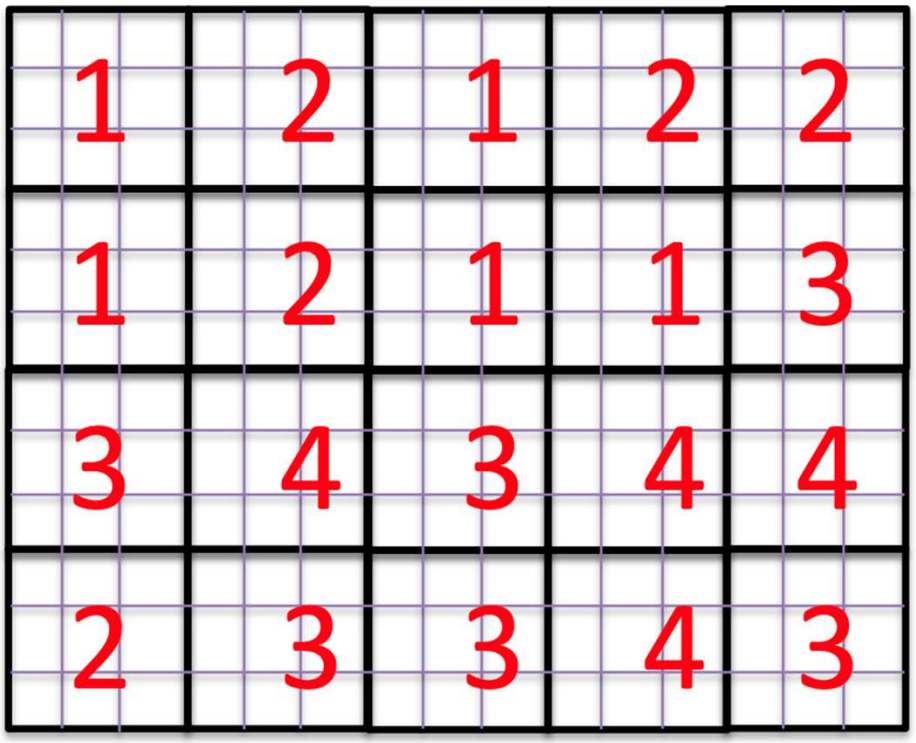
Data Distribution Techniques

- Block cyclic (round robin) distribution

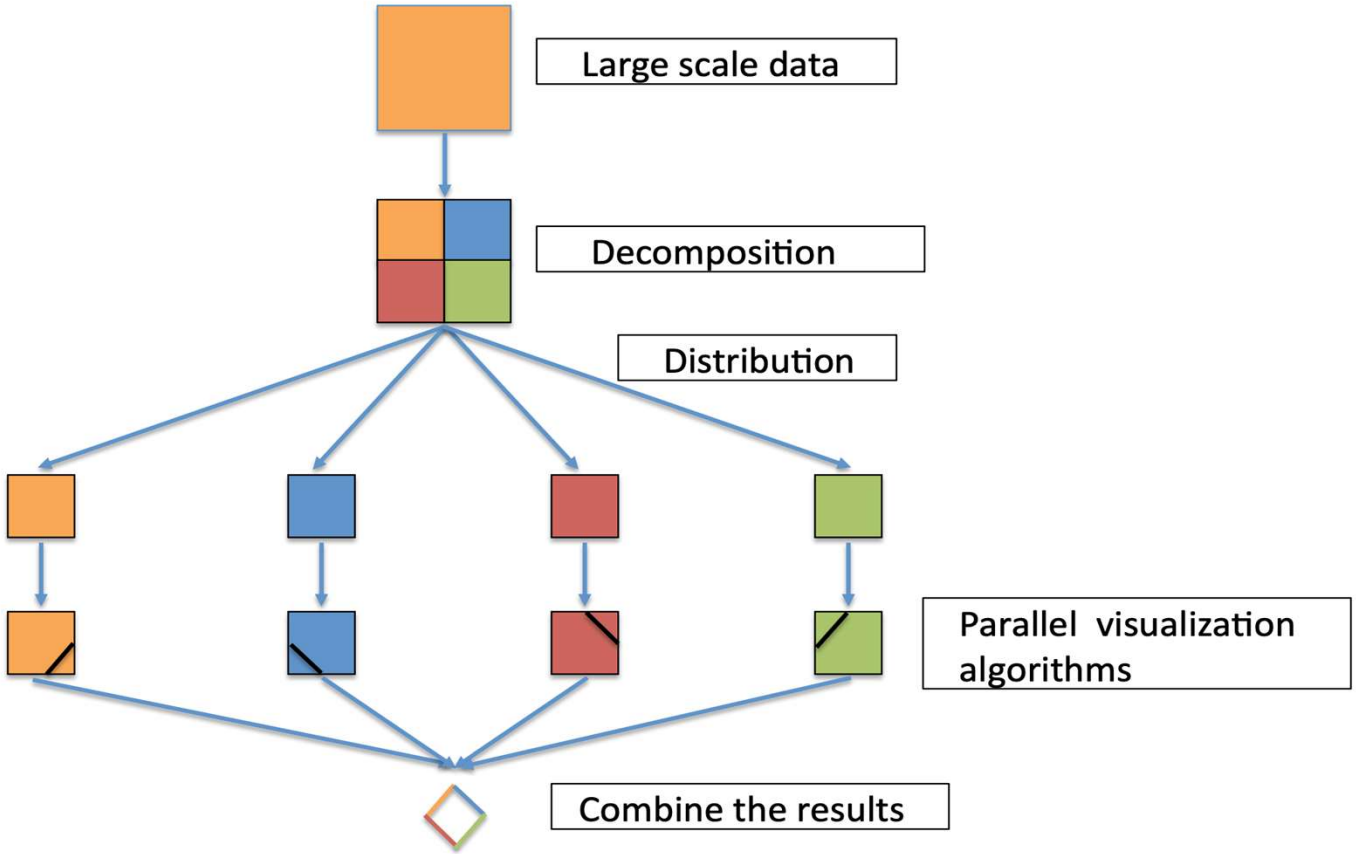


Data Distribution Techniques

- Workload aware

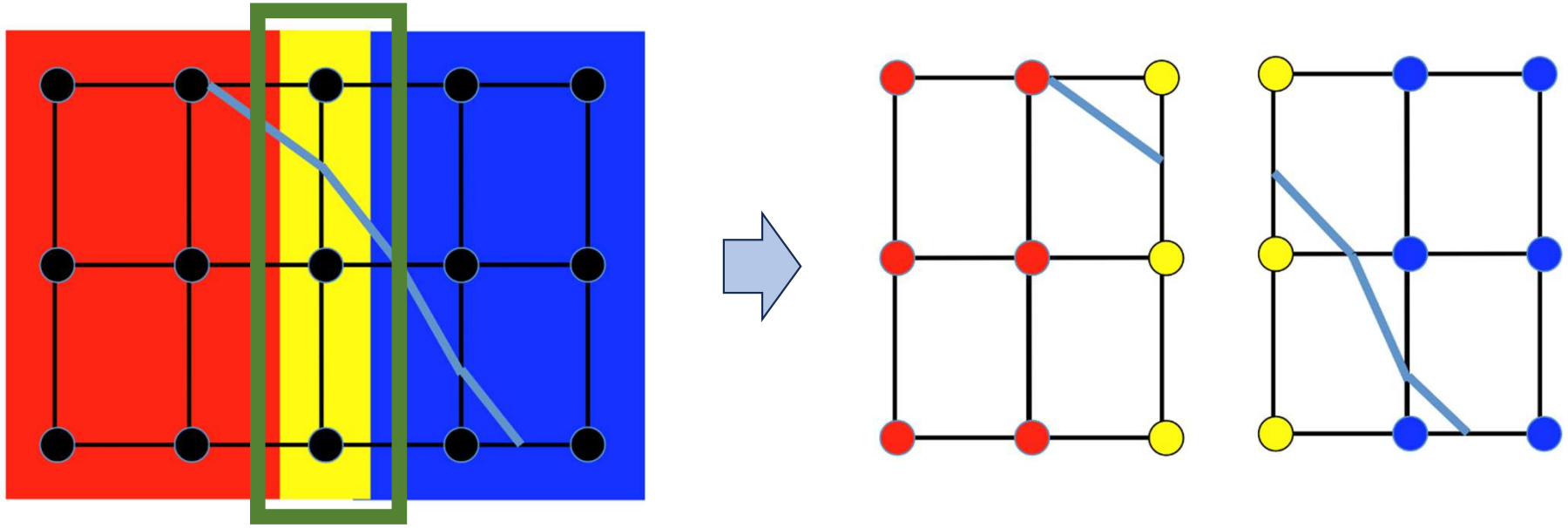


Parallel Data Processing and Aggregation of Partial Results



How to Ensure Continuity at Boundaries

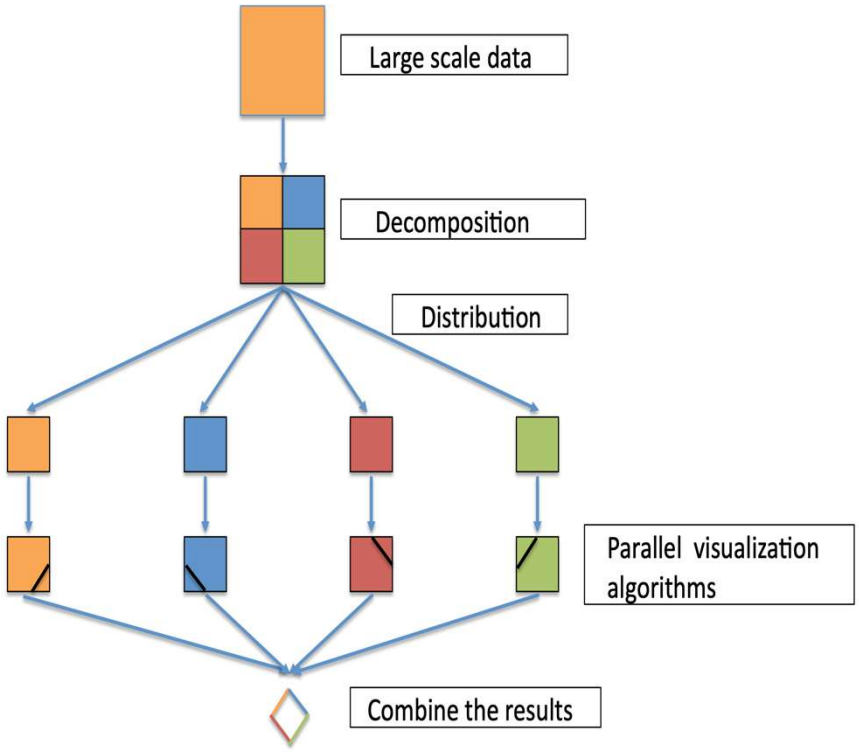
- Need ‘Ghost cells’
  - For interpolation-based tasks, we need to replicate and share boundary data points with neighboring processors
  - Cells in the duplicated layers are called ghost cells



Parallel Isosurface Extraction and Parallel Volume Rendering

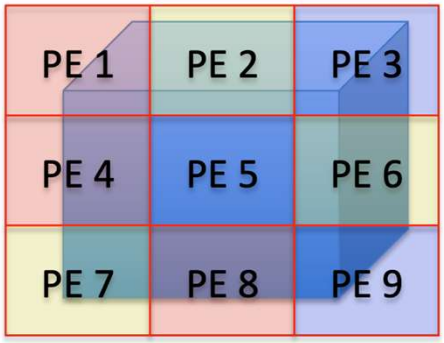
Parallel Isosurface Extraction

- Parallel Marching Cubes (Squares)
- Observation: Each cell can be processed in parallel as there is no dependency among cells
- Steps:
  - Divide the data into chunks of equal size
  - Each processing element (processor) runs Marching Cubes for the cells in the chucks that are assigned to it
  - Return the resulting triangles back to a master node, or render the triangles if parallel rendering is desired



Parallel Volume Rendering

- Parallel Volume Rendering
  - Parallel by pixel – Fits better for Shared memory approach

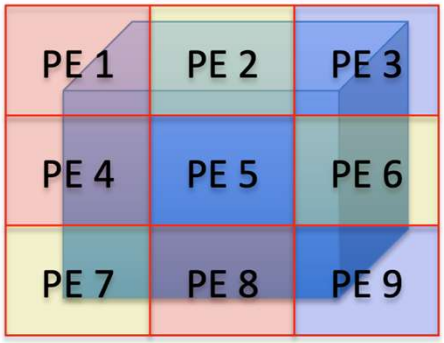


Parallel by pixel

- No image compositing needed
- Little/no communication
- Data often need to be replicated
- Difficult to scale to large data

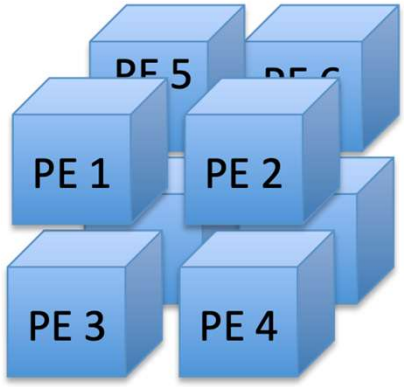
Parallel Volume Rendering

- Parallel Volume Rendering
  - Parallel by pixel – Fits better for Shared memory approach
  - Parallel by data – Fits better for Distributed Memory approach



Parallel by pixel

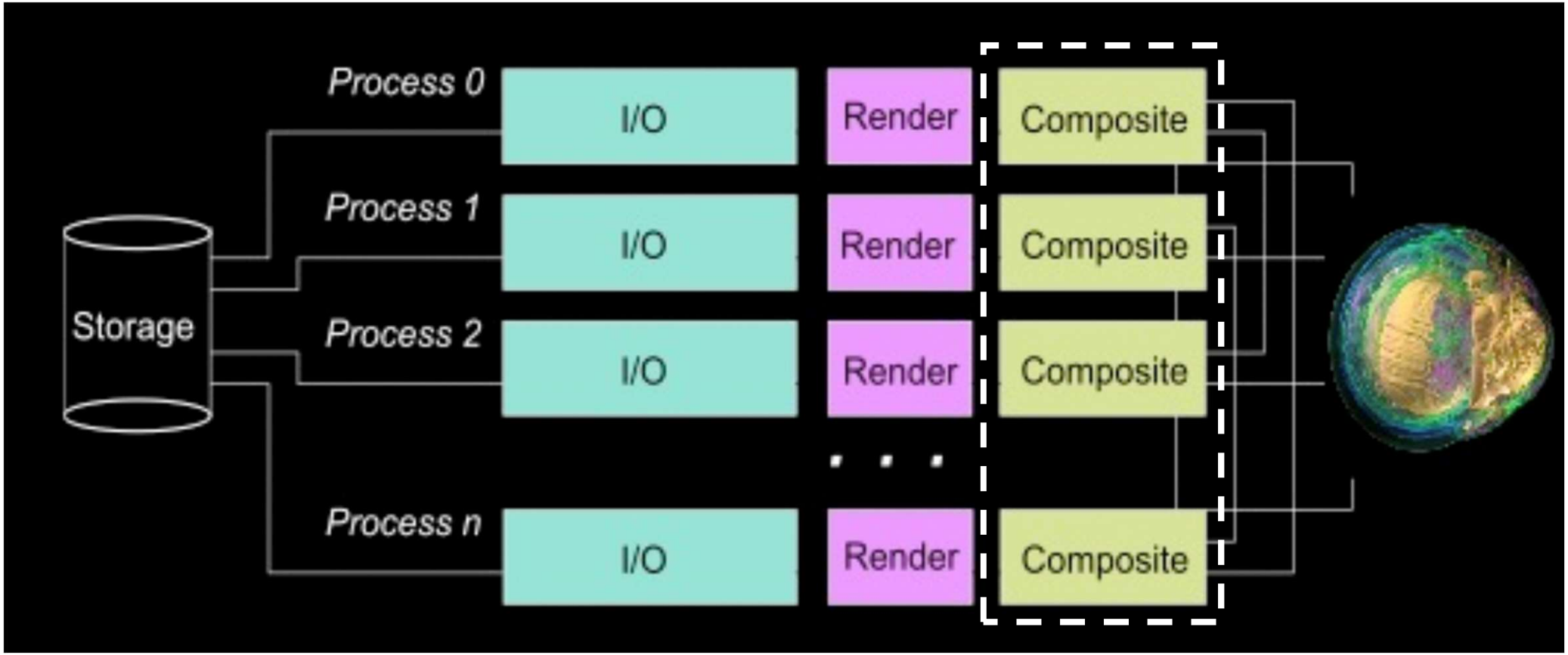
- No image compositing needed
- Data may need to be replicated
- Difficult to scale to large data



Parallel by data

- Image compositing needed
- More communication needed
- Data are distributed
- Easy to scale to large data

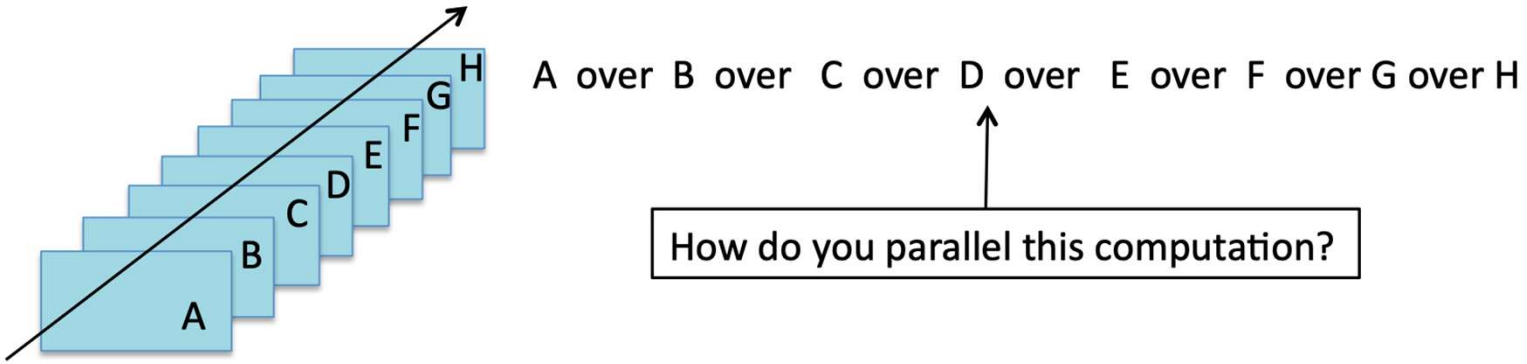
Parallel Volume Rendering



How to do this compositing in parallel efficiently?

Parallel Image Compositing for Volume Rendering

- Composite partial images generated from sub-domains
  - Minimize communication: send and receive images
- Compositing needs to follow the visibility order of ray casting technique



Parallel Image Compositing for Volume Rendering

A over B = AB | C over D = CD | E over F= EF | G over H = GH

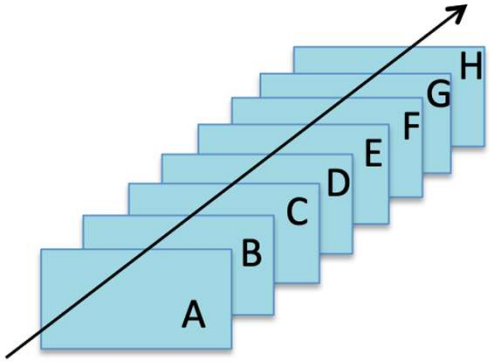
(Requires 4 PEs)

AB over CD = ABCD | EF over GH = EFGH

(Requires 2 PEs)

ABCD over EFGH = ABCDEFGH

(Requires 1 PE)



A over B over C over D over E over F over G over H

How do you parallel this computation?

Parallel Image Compositing for Volume Rendering

A over B = AB | C over D = CD | E over F= EF | G over H = GH

(Requires 4 PEs)

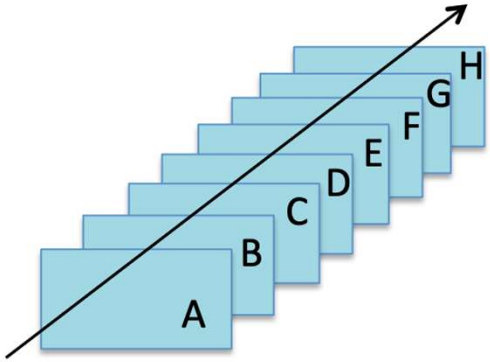
AB over CD = ABCD | EF over GH = EFGH

(Requires 2 PEs)

Load balancing is not good: Half of the processors are busy at every new stage

ABCD over EFGH = ABCDEFGH

(Requires 1 PE)

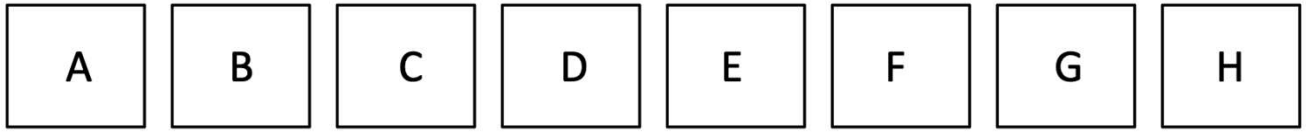


A over B over C over D over E over F over G over H

How do you parallel this computation?

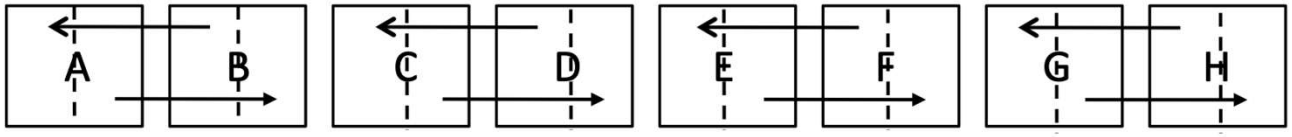
Parallel Image Compositing for Volume Rendering: A better strategy

- keep every processor (PE) busy all the time
- **Recursive halving**



Parallel Image Compositing for Volume Rendering: A better strategy

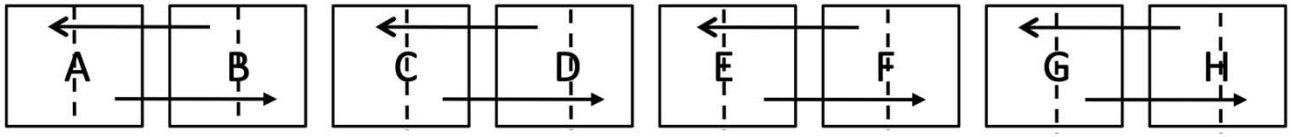
- keep every processor (PE) busy all the time
- Recursive halving



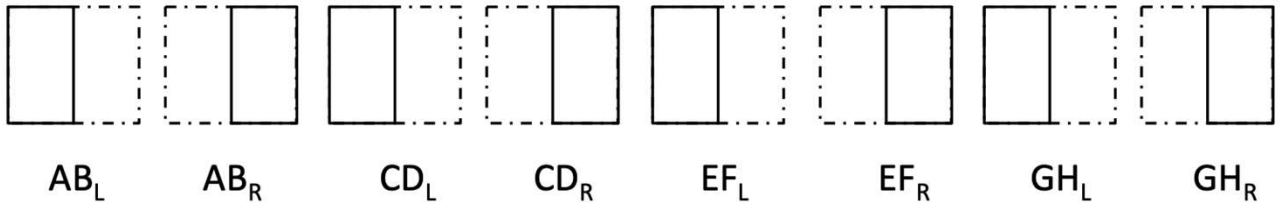
Divide each image into two halves, and give half to the other PE to composite

Parallel Image Compositing for Volume Rendering: A better strategy

- keep every processor (PE) busy all the time
- Recursive halving

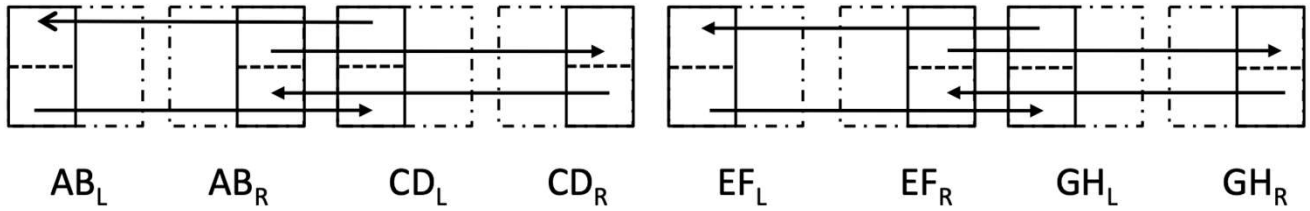


Divide each image into two halves, and give half to the other PE to composite



Parallel Image Compositing for Volume Rendering: A better strategy

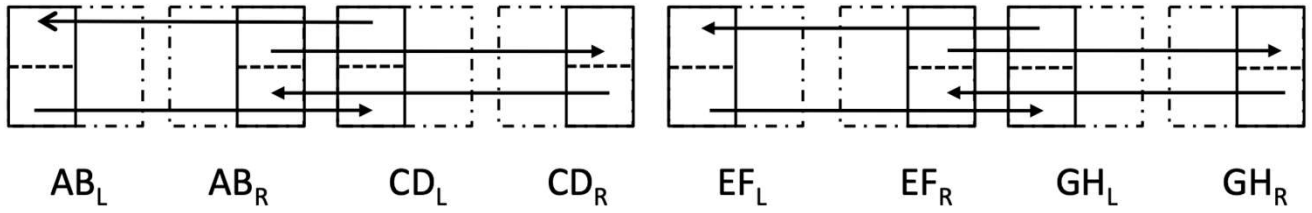
- keep every processor (PE) busy all the time
- Recursive halving



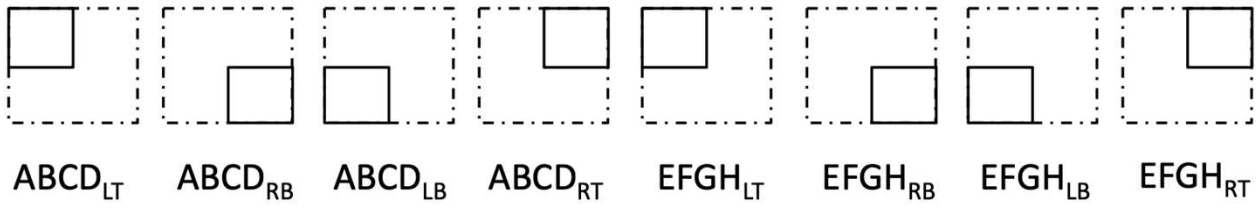
Divide the half image further into top and bottom halves, and give it to the corresponding PE of to composite

Parallel Image Compositing for Volume Rendering: A better strategy

- keep every processor (PE) busy all the time
- Recursive halving

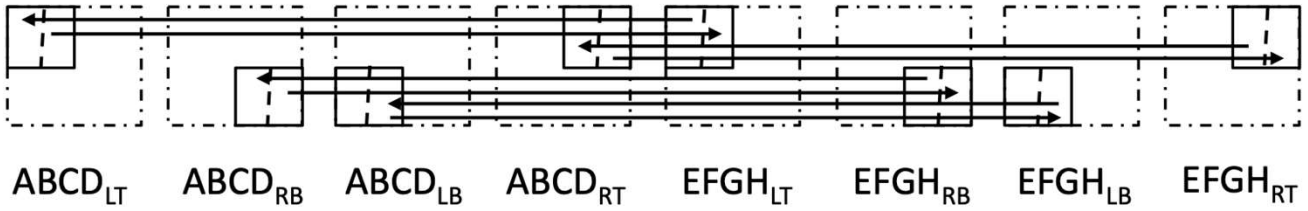


Divide the half image further into top and bottom halves, and give it to the corresponding PE of to composite



Parallel Image Compositing for Volume Rendering: A better strategy

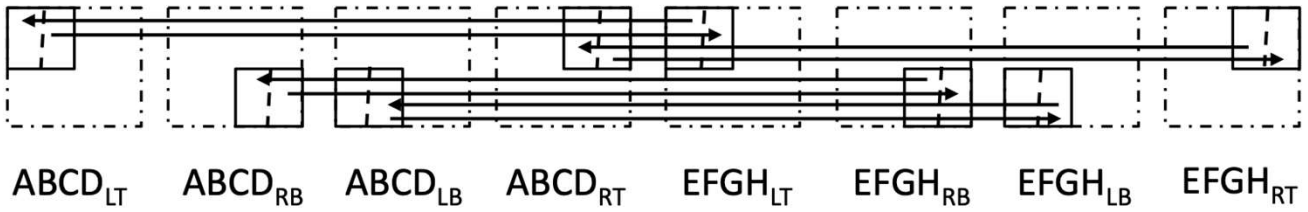
- keep every processor (PE) busy all the time
- Recursive halving



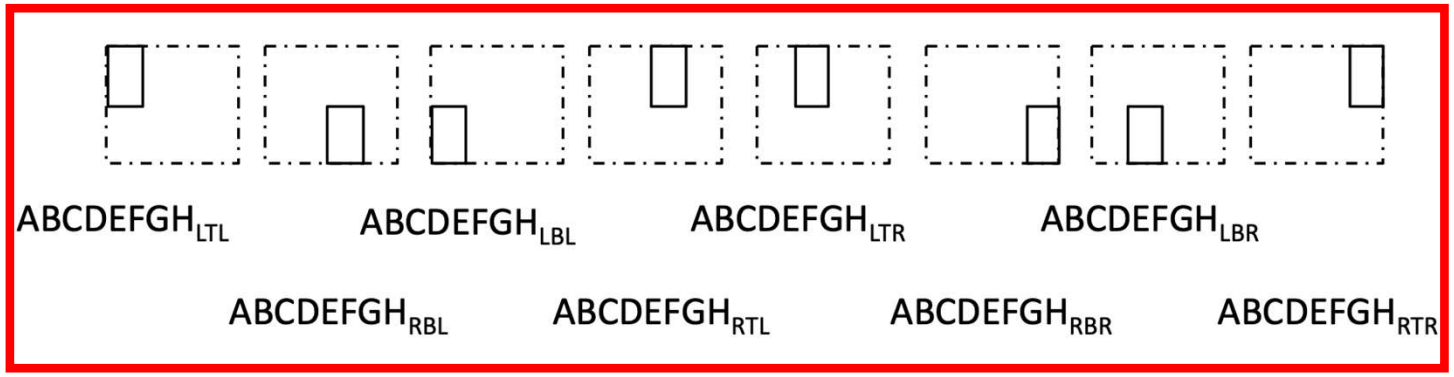
Divide the 1/4 image further into left and right halves, and give it to the corresponding PE of to composite

Parallel Image Compositing for Volume Rendering: A better strategy

- keep every processor (PE) busy all the time
- Recursive halving

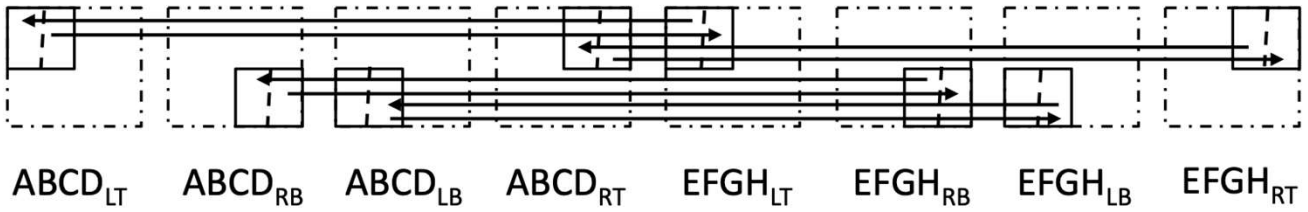


Divide the 1/4 image further into left and right halves, and give it to the corresponding PE of to composite



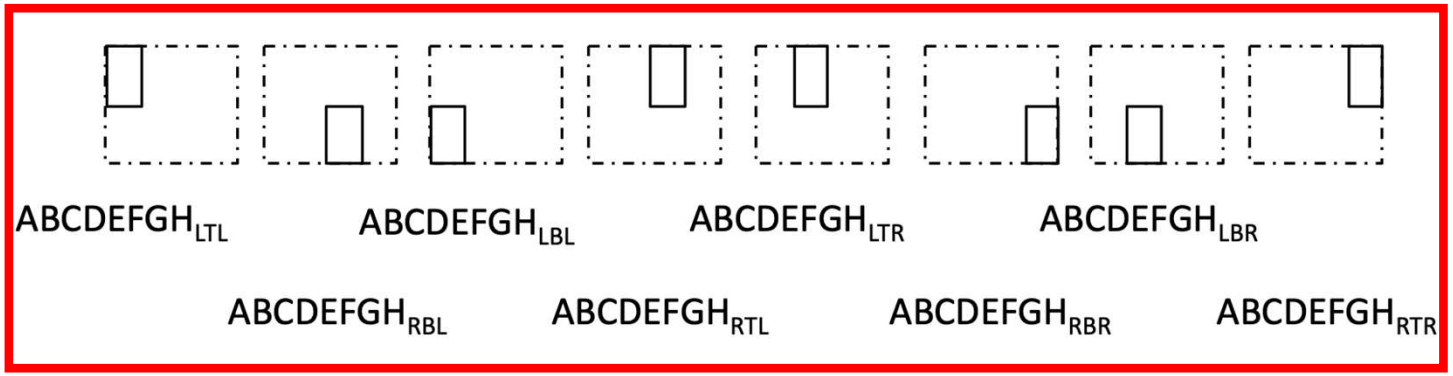
Parallel Image Compositing for Volume Rendering: A better strategy

- keep every processor (PE) busy all the time
- Recursive halving



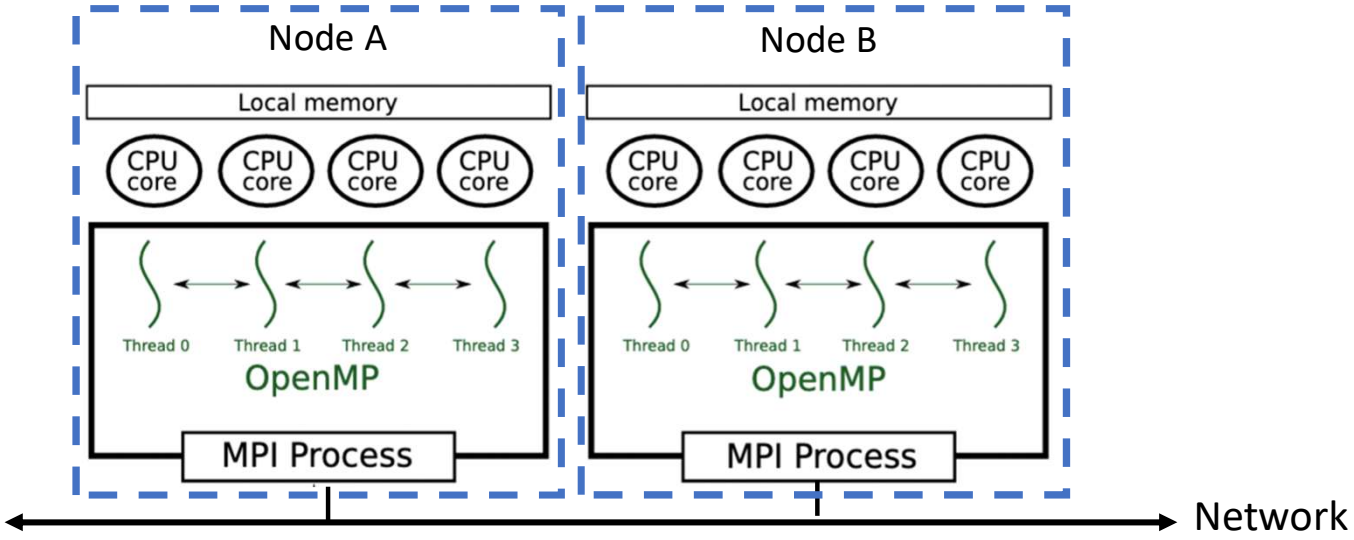
Divide the 1/4 image further into left and right halves, and give it to the corresponding PE of to composite

Load balanced approach: all workers are busy with equal workload at all the time



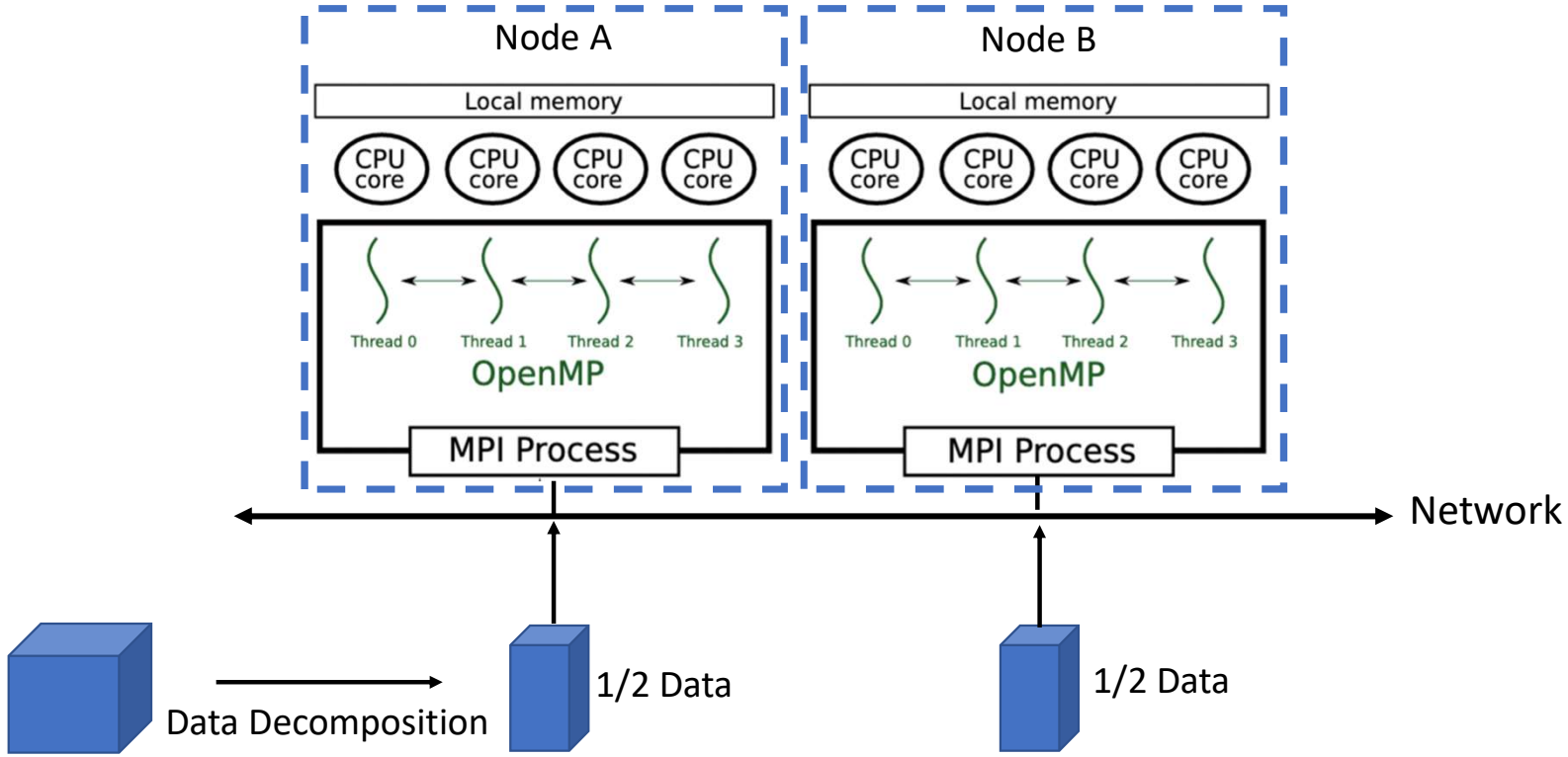
Hybrid Parallel Volume Rendering

- Use MPI + OpenMP together to extract fine grained parallelism for very large-scale data set



Hybrid Parallel Volume Rendering

- Use MPI + OpenMP together to extract fine grained parallelism for very large-scale data set



Hybrid Parallel Volume Rendering

- Use MPI + OpenMP together to extract fine grained parallelism for very large-scale data set