Parallelizing Fundamental Algorithms such as Sorting on Multi-core Processors for EDA Acceleration

Masato Edahiro

NEC Corporation
University of Tokyo

Empowered by Innovation NEC

Outline

- Multi-core, Everywhere
  - Scalable Algorithms
  - How to get Performance?
- Example of Scalable Algorithms: Sorting
- Experiments
- Future Directions

Multi-Core, Everywhere

- Single CPU: Limit of Performance Scaling
  - Device Minitualization Continues.
  - BUT SELECT "Low Power" OR "High Frequency"
- Multi-Core, Everywhere
  - Severe Power Limitation
  - Multiple Low-Power CPUs
    - Servers, Personal Computers, High-end Embedded Systems
- Software: No More Performance Scaling WITHOUT PARALLELISM

Types of Parallelism

- Distribute Multiple Applications (in systems) on Multi-Core
- Accelerate Single Application on Multi-Core
  - Algorithms SHOULD BE SCALABLE
  - Even if an Algorithm is Scalable, in Many Cases, its Implementation is NOT Scalable in Performance

Today's Topic

What is Scalable Algorithm?

1. ***P-times Smaller Time Complexity with P CPUs***

2. ***Comparable Processing Time on a CPU*** **compared with optimized algorithms for single-CPU processors**

3. ***Higher Speed-Up with Multiple CPUs***

Previous Work

- Has been done on
  - Scientific Calculation (for Servers)
  - Media Processing (for SIMD Machines, e.g. GPU)

- EDA Acceleration
  - Easy to Parallelize: Meta Heuristics
    - Simulated Annealing, Genetic Algorithms, Neural Network, etc.
  - Difficult to Parallelize
    - Basic Algorithms: e.g. Sorting
    - Graph & Network Algorithms: e.g. Tree Search

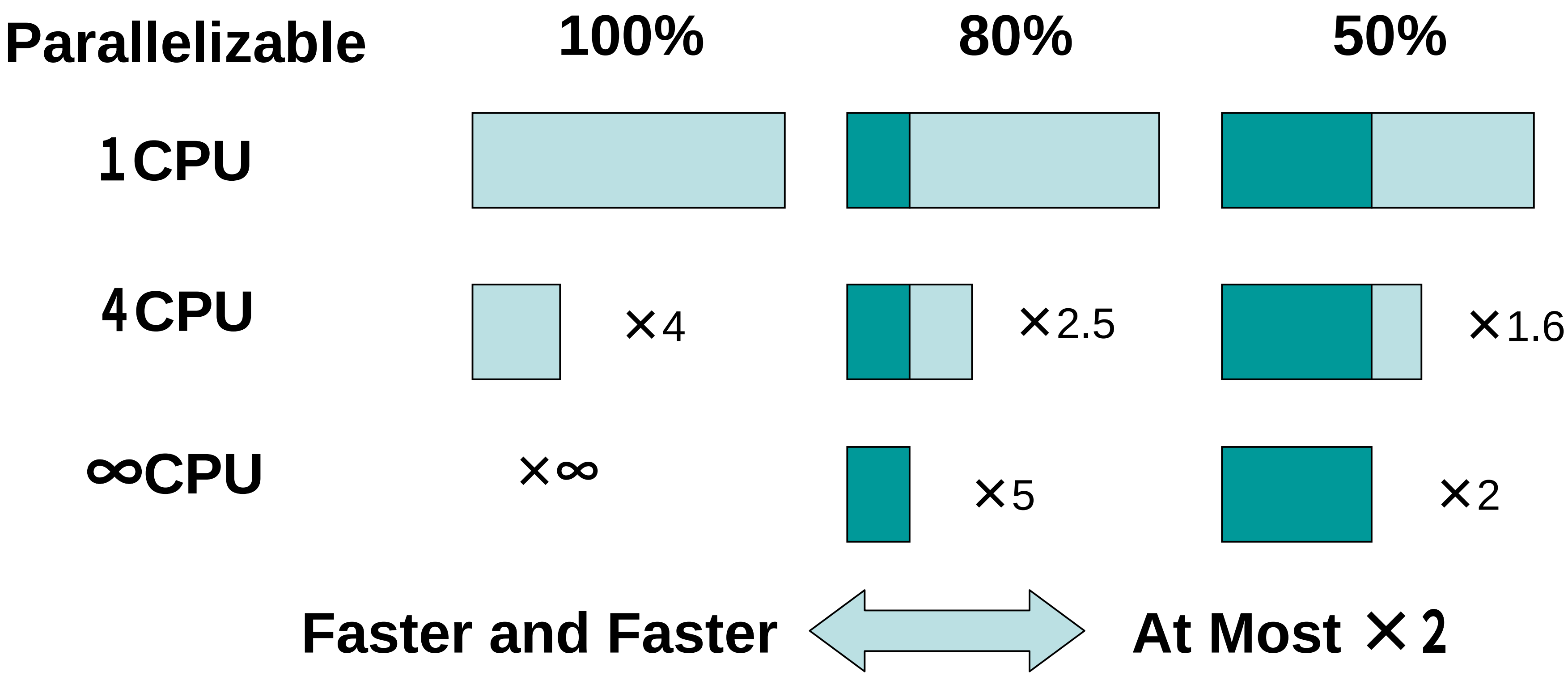
Types of Parallelism

- Distribute Multiple Applications (in systems) on Multi-Core
- Accelerate Single Application on Multi-Core
  - Algorithms SHOULD BE SCALABLE
  - Even if an Algorithm is Scalable, in Many Cases, its Implementation is NOT Scalable in Performance

Need to Care About for Performance Scalability

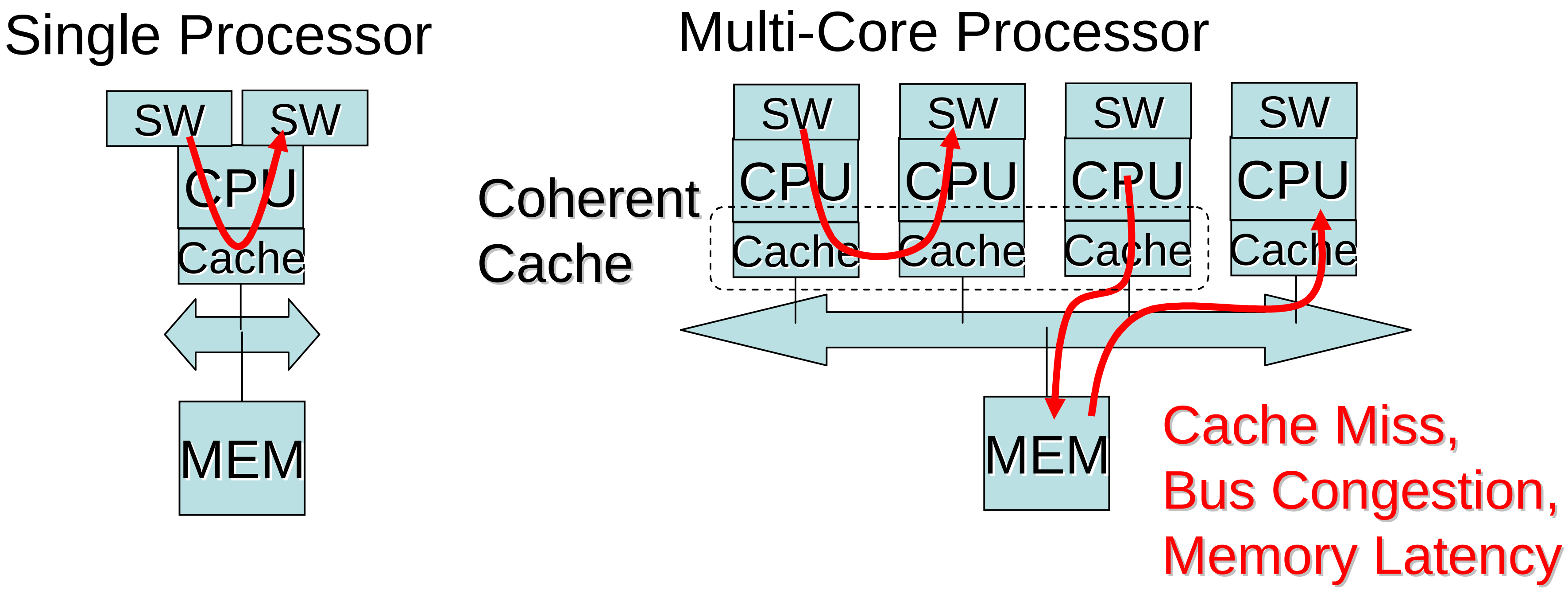
- Amdahl's Law
- Memory Access
- Inter-Core Communication
- Granularity
- Load Balancing

Amdahl's Law


Parallelizable
100%
80%
50%
1CPU
4CPU
×4
×2.5
×1.6
∞CPU
×∞
×5
×2
Faster and Faster
At Most ×2

Memory Access & Inter-Core Communication

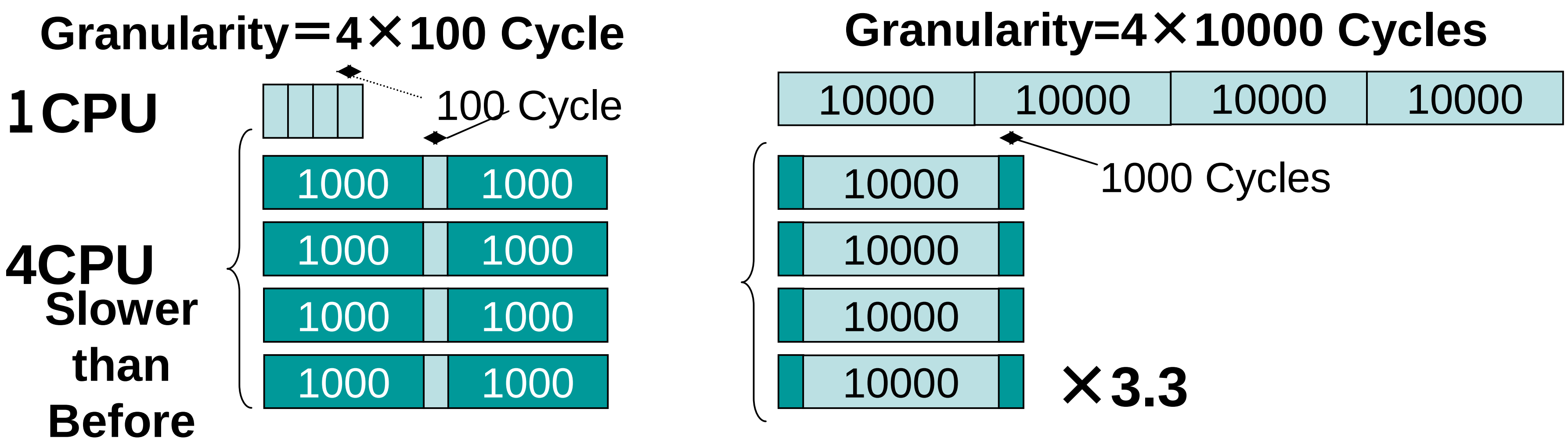
- **Memory Access and Inter-Core Communication (using Shared Memory) is often critical for Performance**



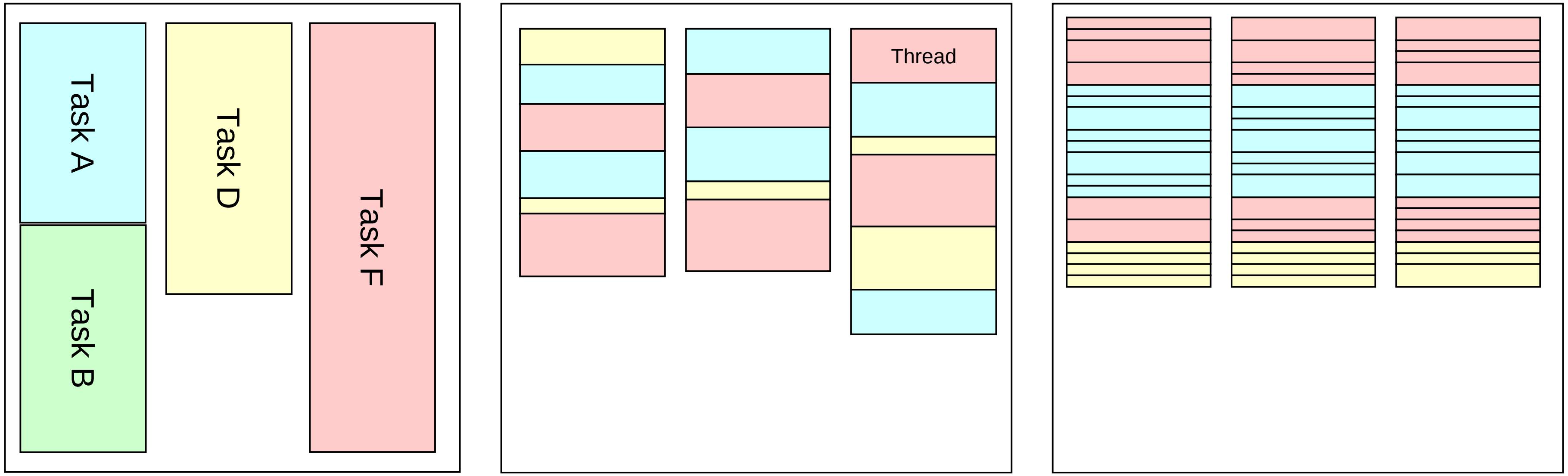
Granularity

- **Overhead of Parallelism**
  - Library Functions of Parallel Programming have some Overhead Because of Memory Access and Inter-CPU Communication
  - Granularity of Parallel Programming is Important

Example: Overhead = 1000 Cycles



Load Balancing



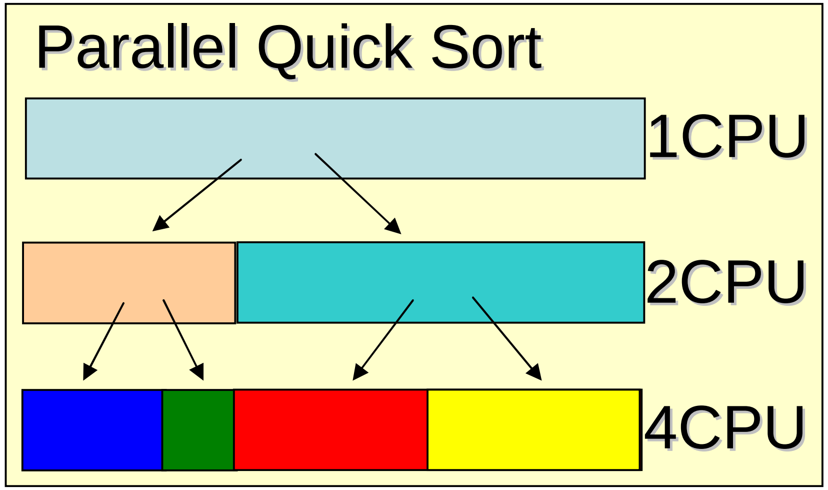
- **Granularity vs. Load Balancing**
  - **Fine Grain: Better Load Balancing**
  - **Coarse Grain: Smaller Overhead**

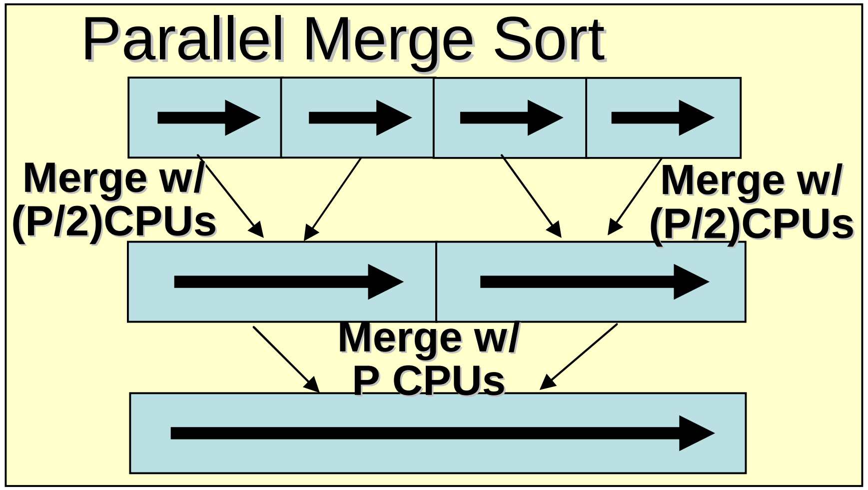
Outline

- Multi-core, Everywhere
  – Scalable Algorithms
  – How to get Performance?
- Example of Scalable Algorithms: Sorting
- Experiments
- Future Directions

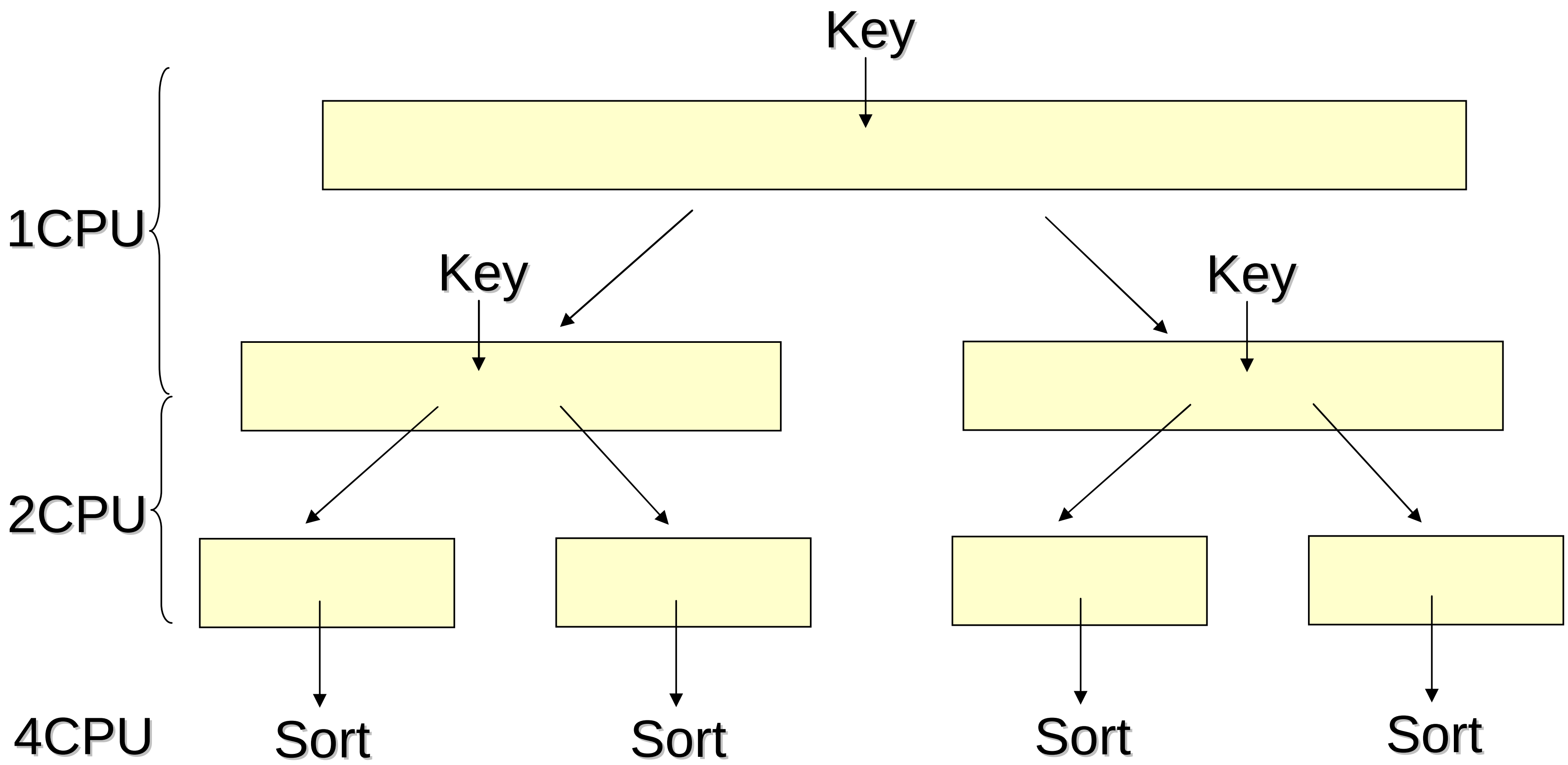
Sorting

- Most Fundamental, Frequently Used
  - should be
    As Fast As Possible
- Need More Scalability
  - Parallel Quick Sort
    - Difficult to Parallelize in First Several Recursions
  - Parallel Merge Sort
    - More Copy
      (Slower than Quick Sort on Single Processor)
    - Merge with Multiple CPUs is Complicated.
  - Other Parallel Sorting
    - Slow with small # of CPUs



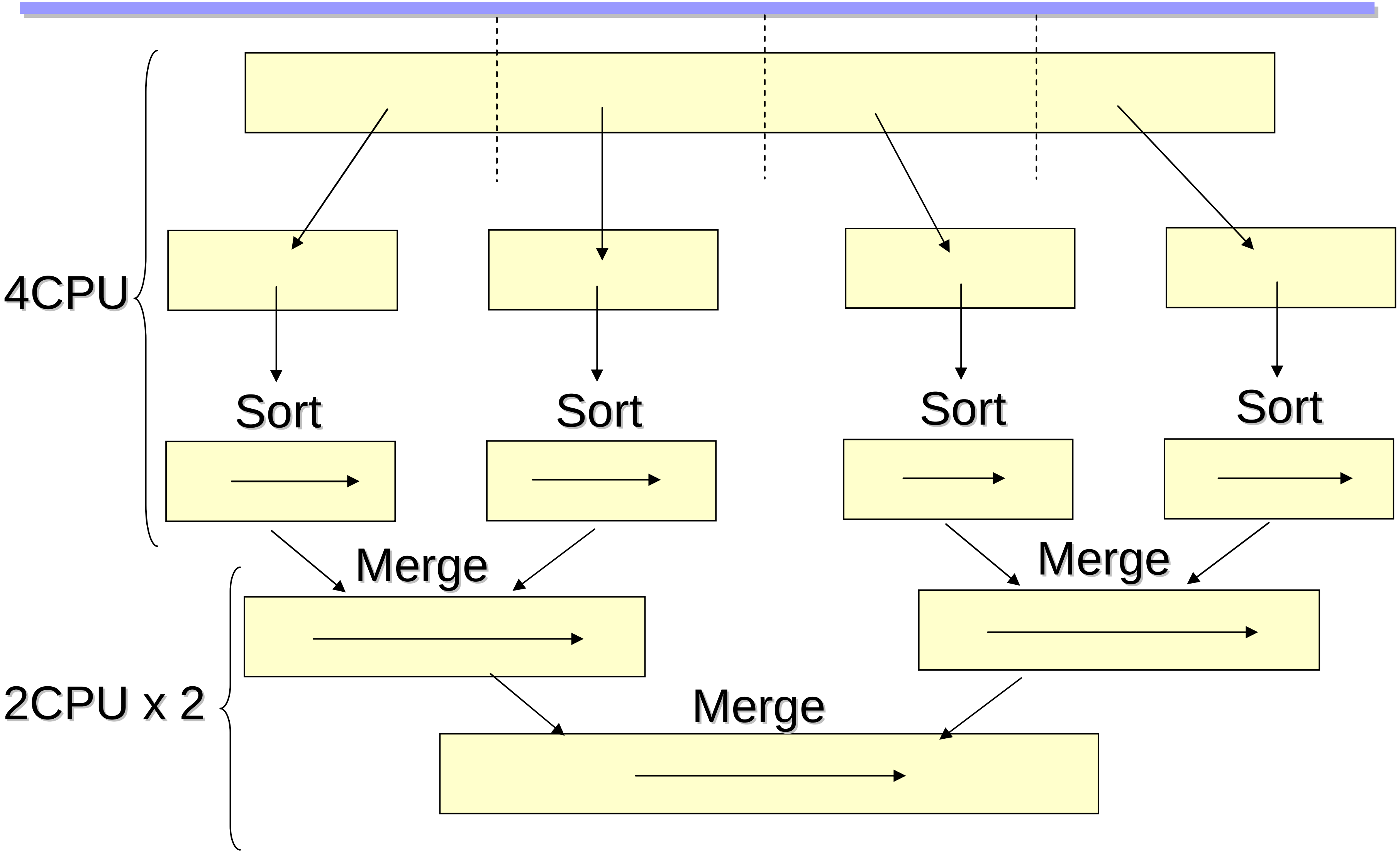


Parallel Quick Sort



- Not Easy to Utilize Multi-Core in First Several Steps
- Often Makes Load Imbalance

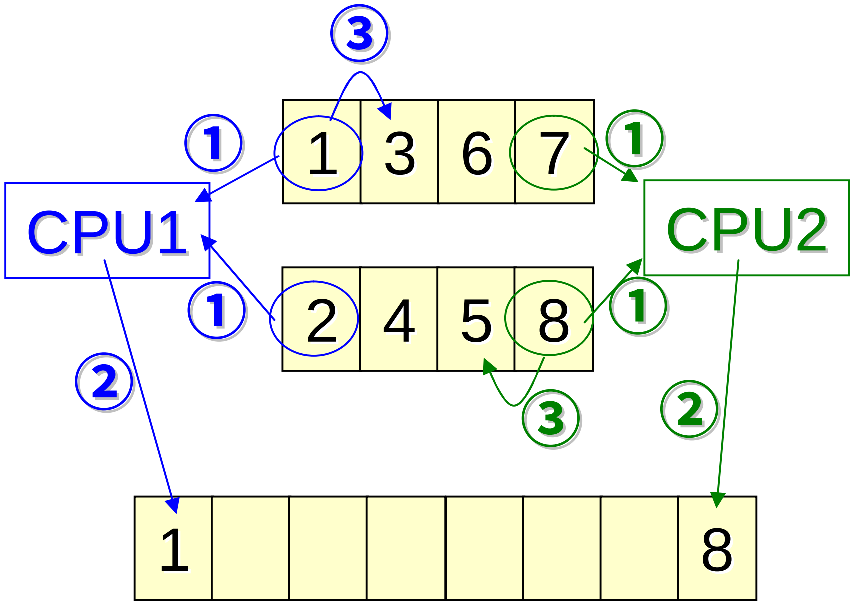
Parallel Merge Sort (1)



- Very Good Load Balancing

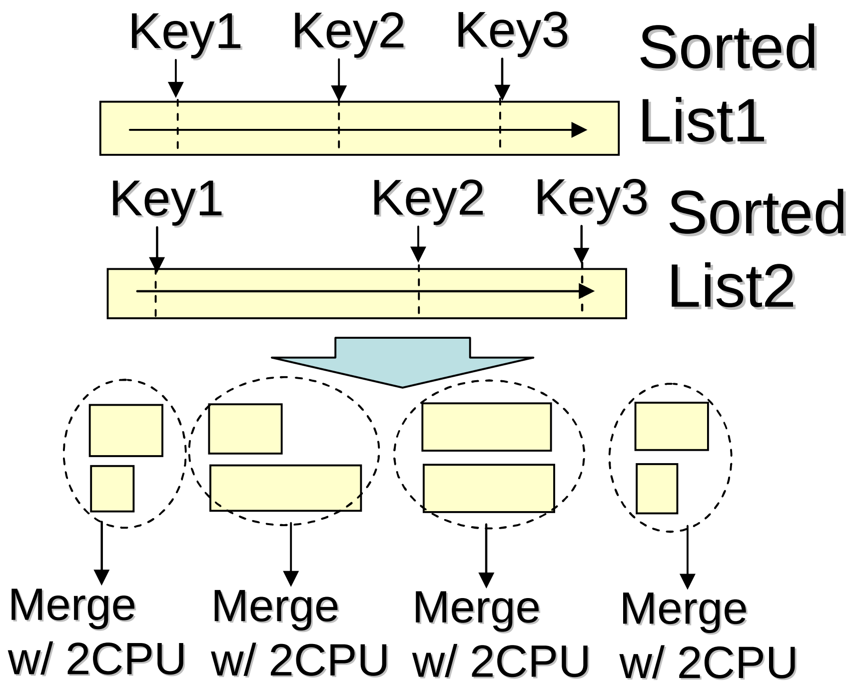
Parallel Merge Sort (2)

- Merge with 2CPUs

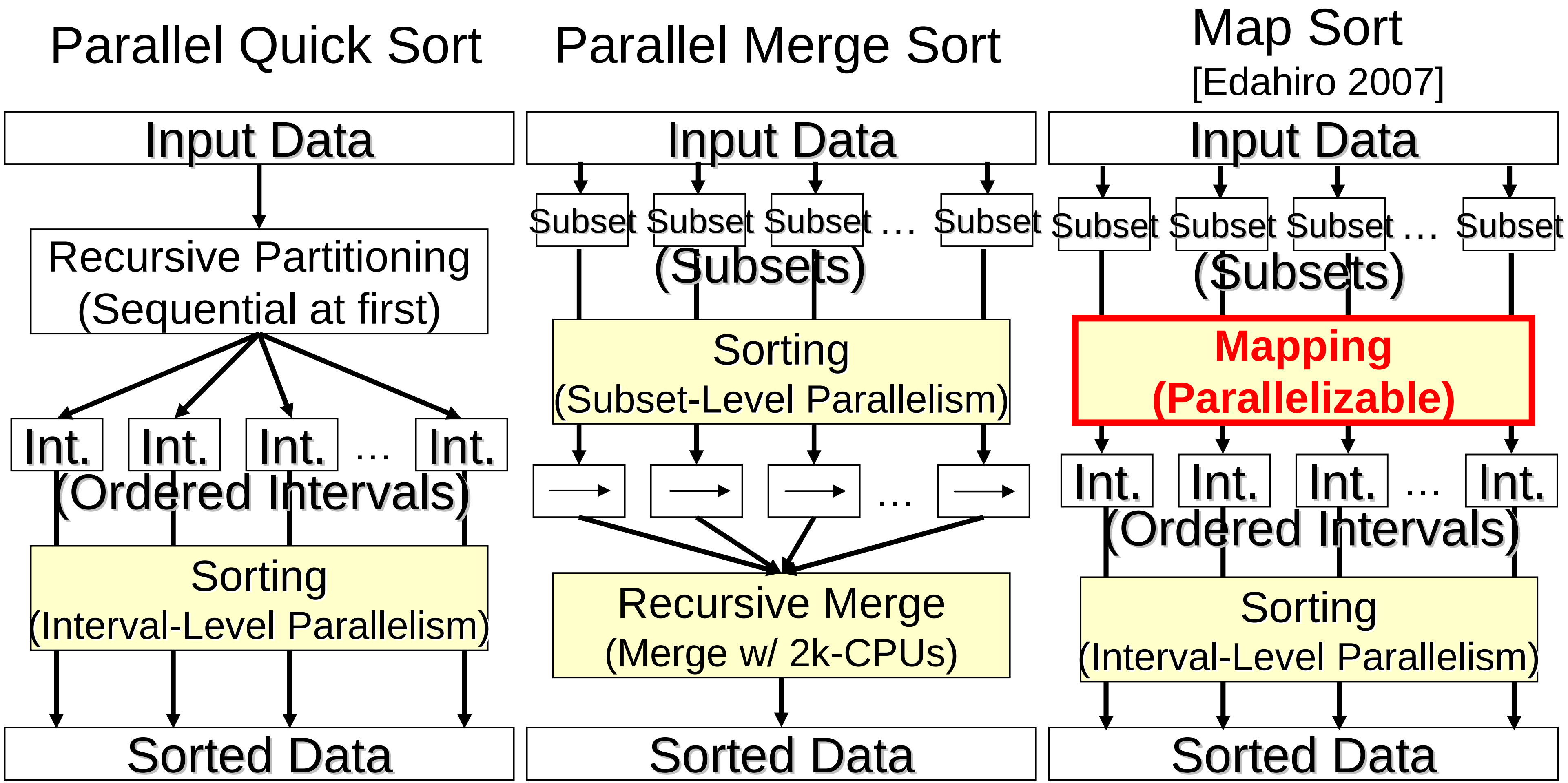


- Need More Data Copy in Merging
- May Cause Load Imbalance (in Key Selection)

- Merge with 2k-CPUs
  1. Select k-1 keys
  2. Divide a pair of sorted lists into k pairs of sorted lists
  3. Merge each pair with 2CPUs



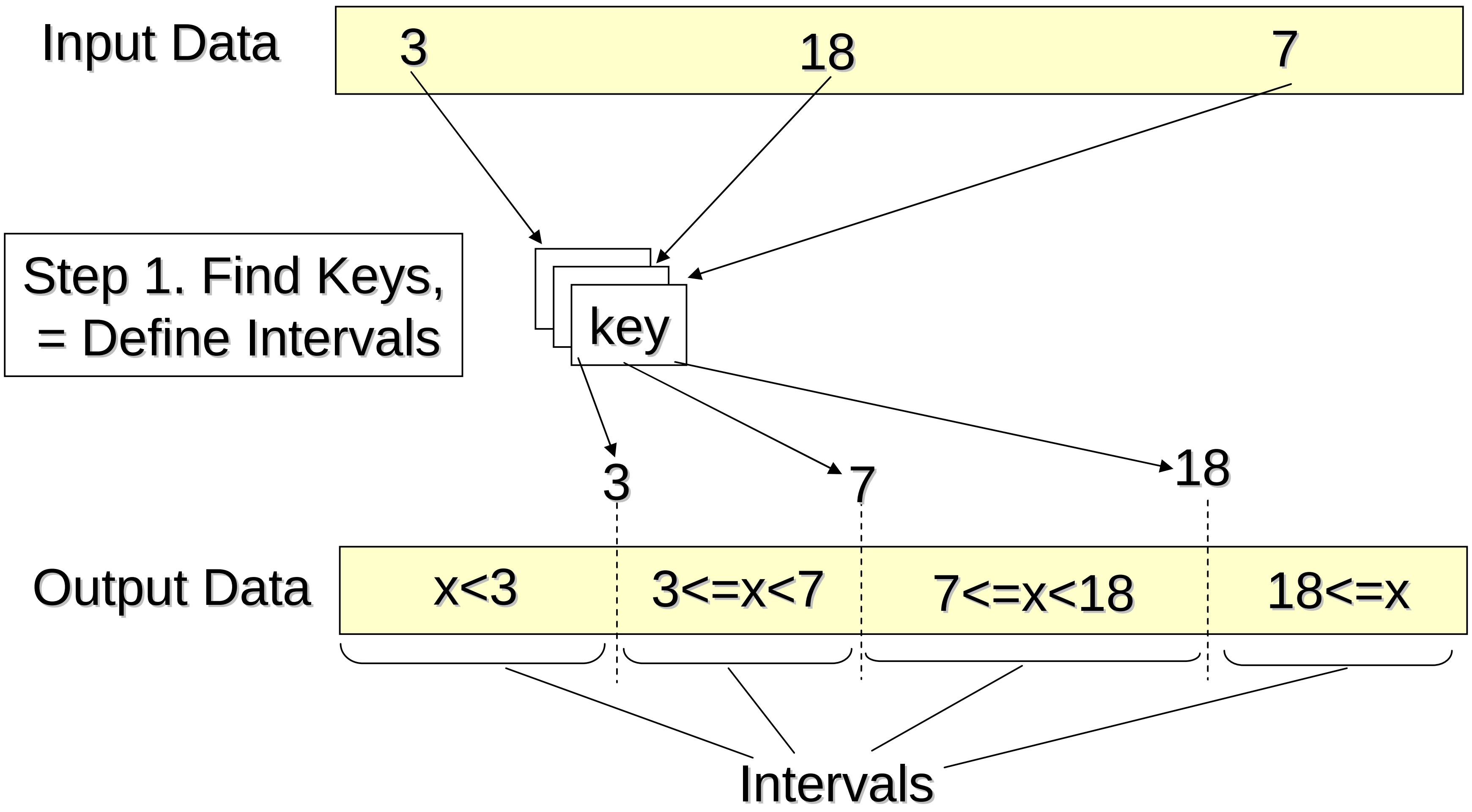
Map Sort and Previous Algorithms



- Sequential Part
- Load Imbalance

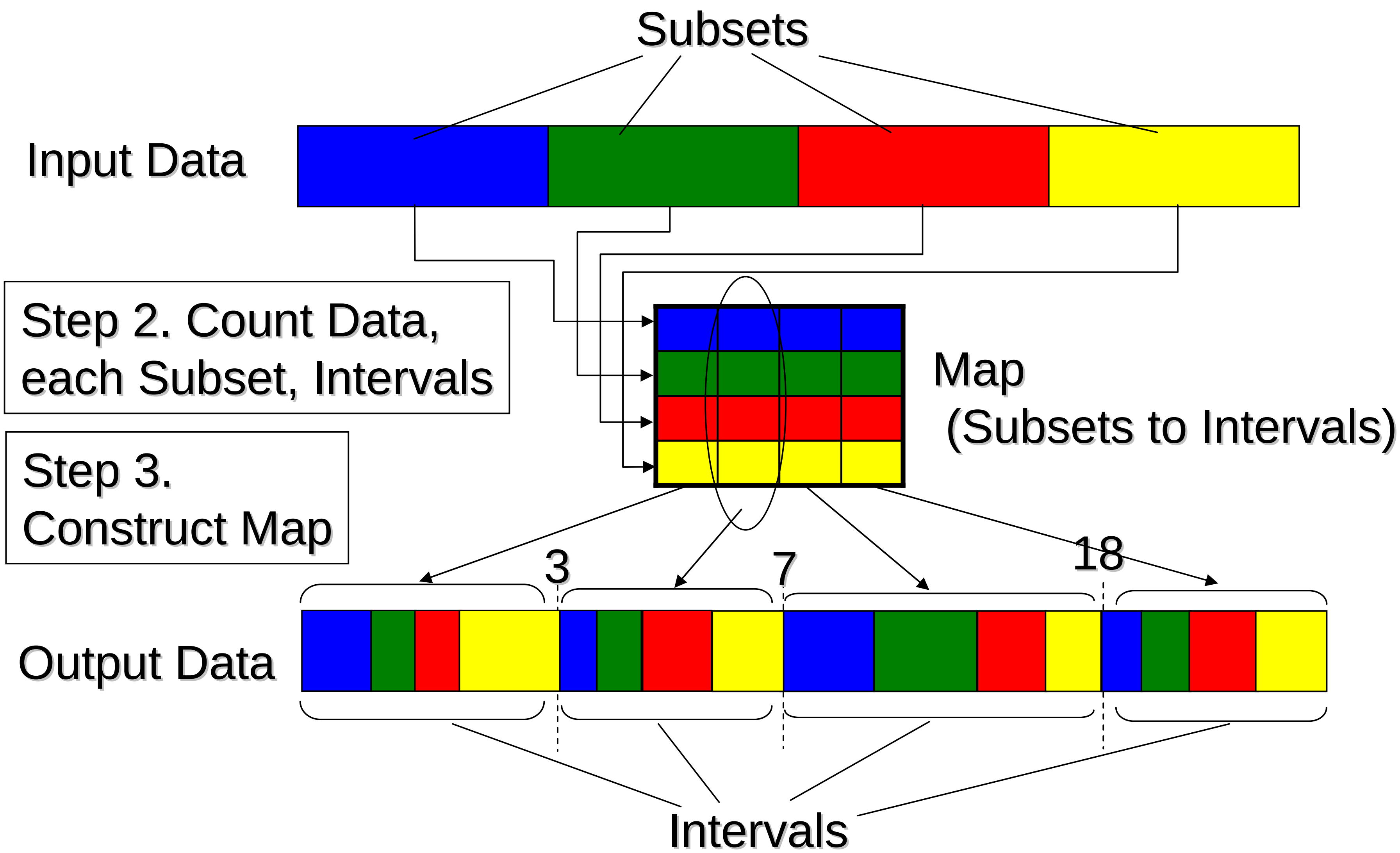
- Good Load Balance
- More Copies

Map Sort(1)



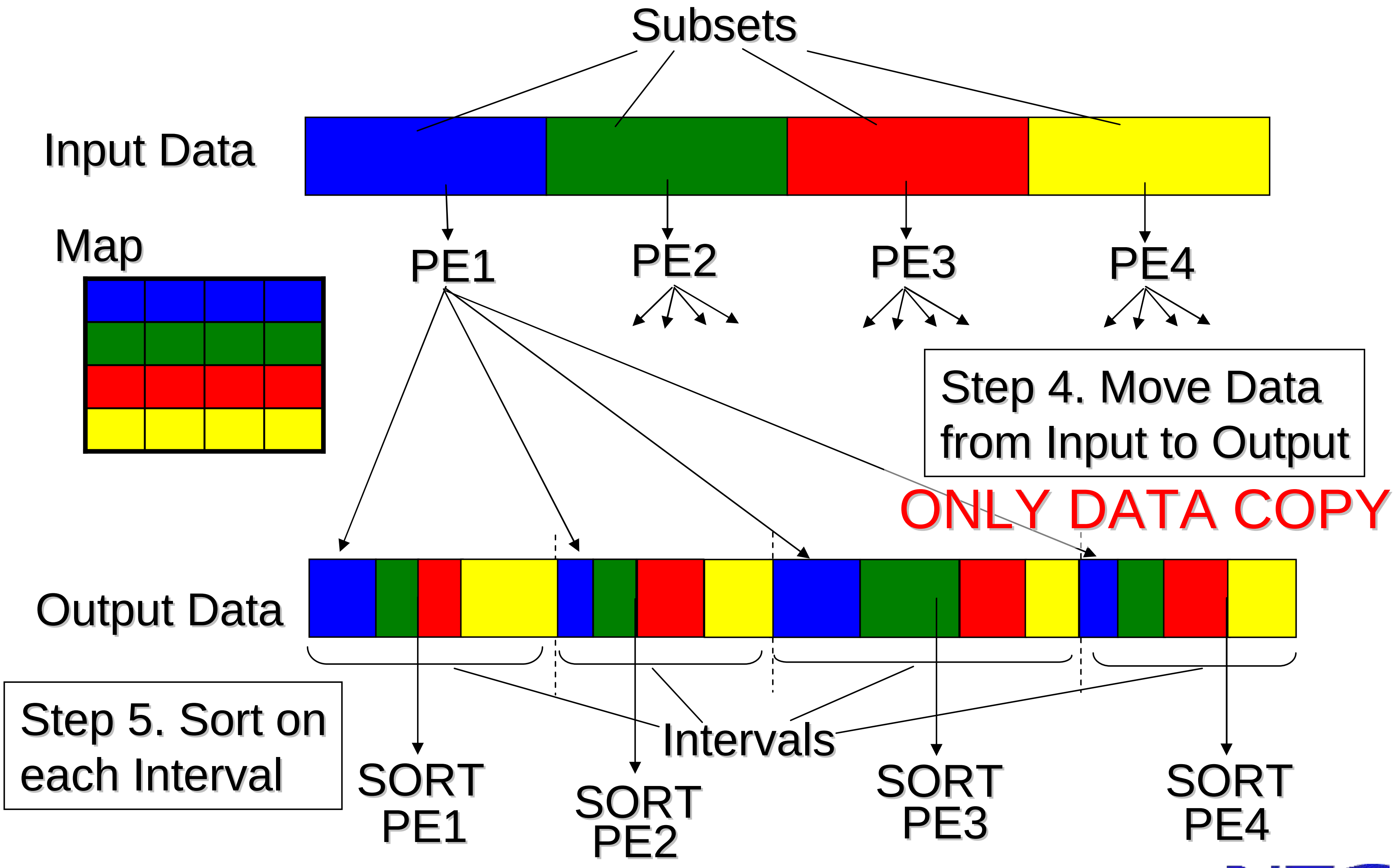
Input Data
3
18
7
Step 1. Find Keys,
= Define Intervals
key
3
7
18
Output Data
x<3
3<=x<7
7<=x<18
18<=x
Intervals

Map Sort(2)

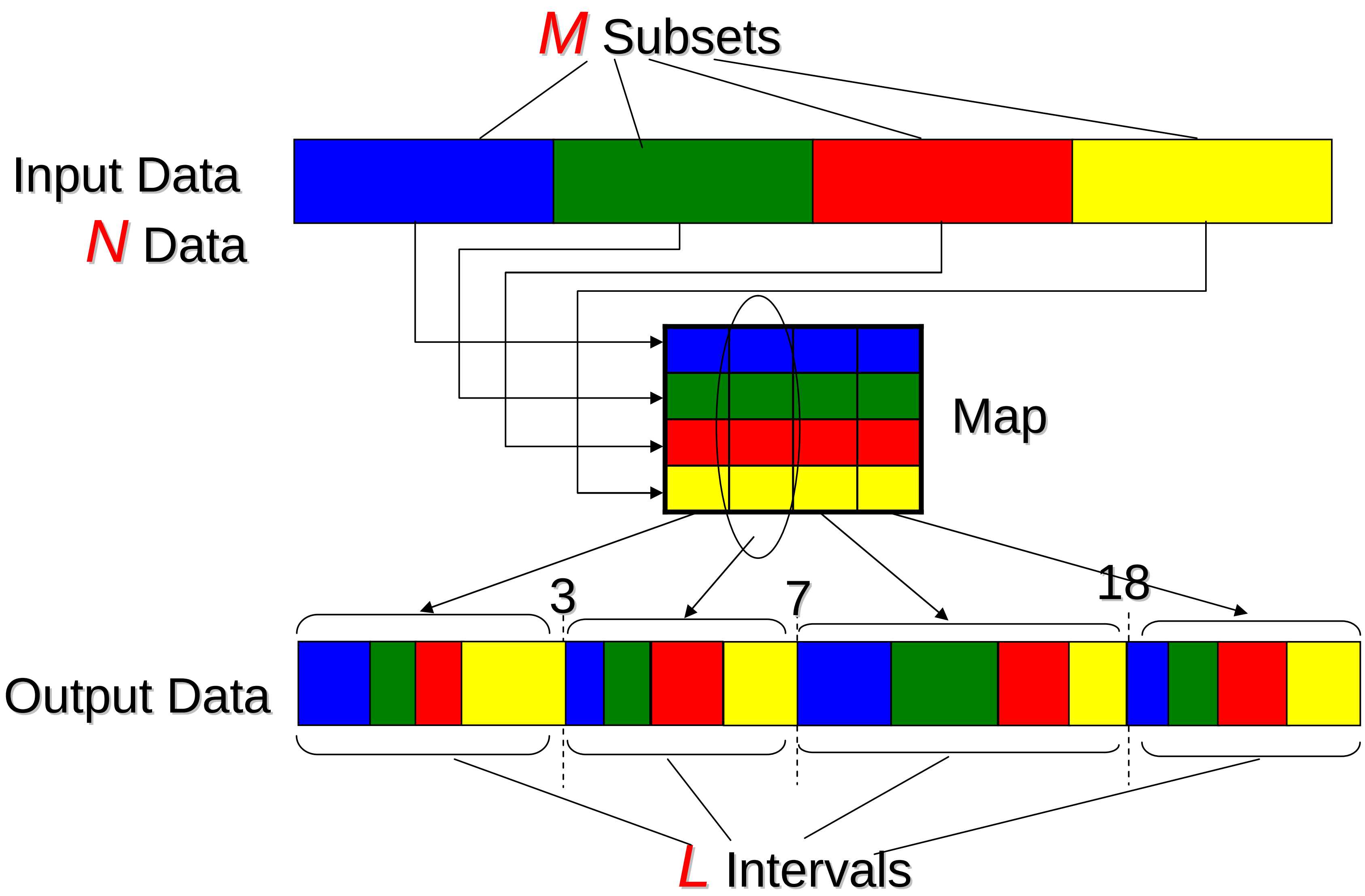


Subsets
Input Data
Step 2. Count Data, each Subset, Intervals
Step 3. Construct Map
Map (Subsets to Intervals)
3
7
18
Output Data
Intervals

Map Sort(3)


Subsets
Input Data
Map
PE1
PE2
PE3
PE4
Step 4. Move Data from Input to Output
ONLY DATA COPY
Output Data
Step 5. Sort on each Interval
Intervals
SORT PE1
SORT PE2
SORT PE3
SORT PE4

Map Sort - - - Parameters

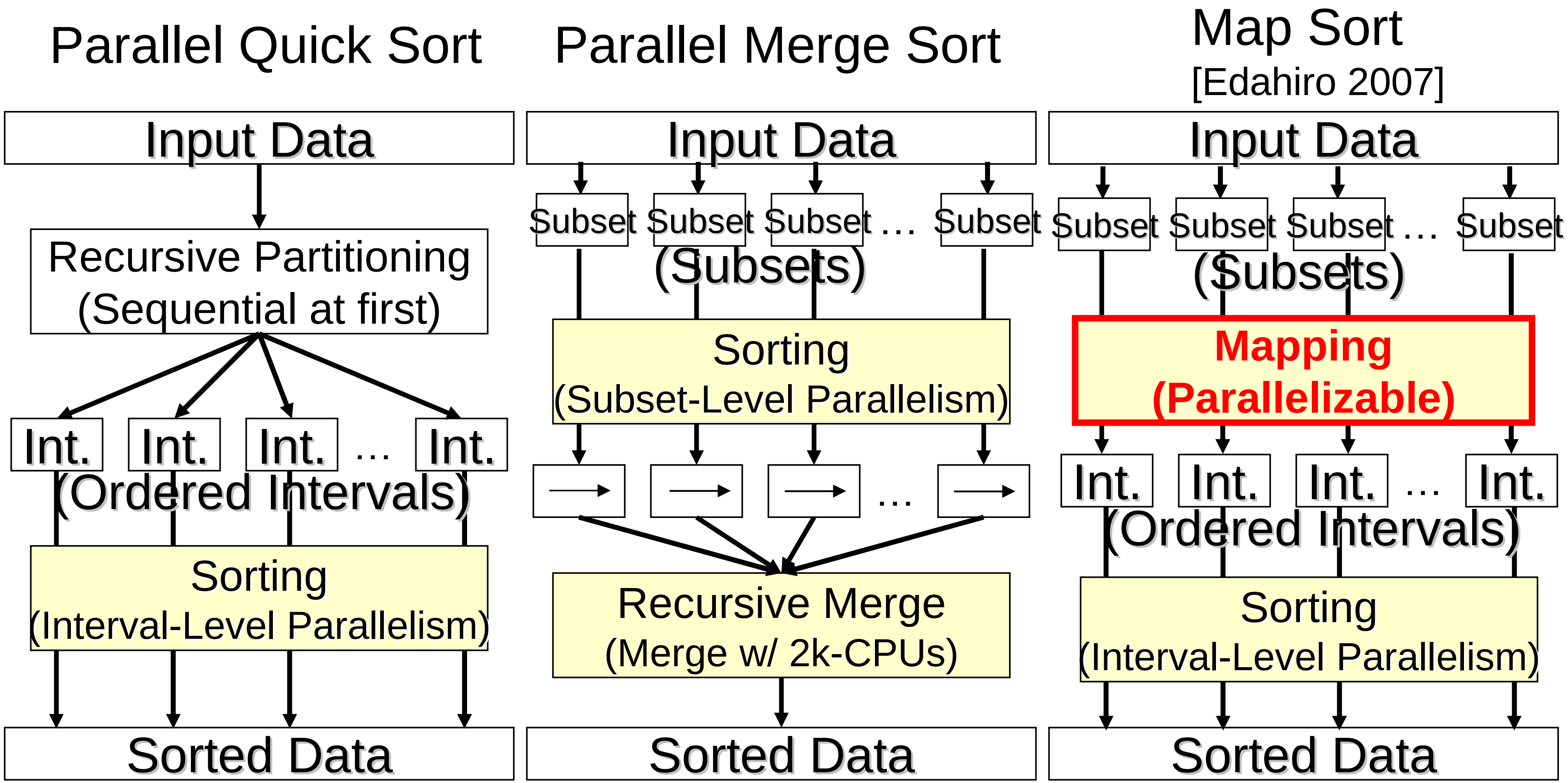


Complexity (Map Sort)

- Assuming L, $M=O(P)$, P: # of CPUs
- Space Complexity: $O(N+P^2)$
  - Output Array: $O(N)$, Map: $O(P^2)$
- Time Complexity: $O((N/P) \log N)$ with P CPUs
  - Find Keys: $O(P)$
  - Count Data: $O((N/P) \log P)$
  - Construct Map: $O(P)$
  - Move Data from Input to Output: $O(N/P)$
  - Sort on Intervals:

$$O((N/P) \log (N/P)) = O((N/P) \log N)$$

Map Sort and Previous Algorithms



- Sequential Part
- Load Imbalance

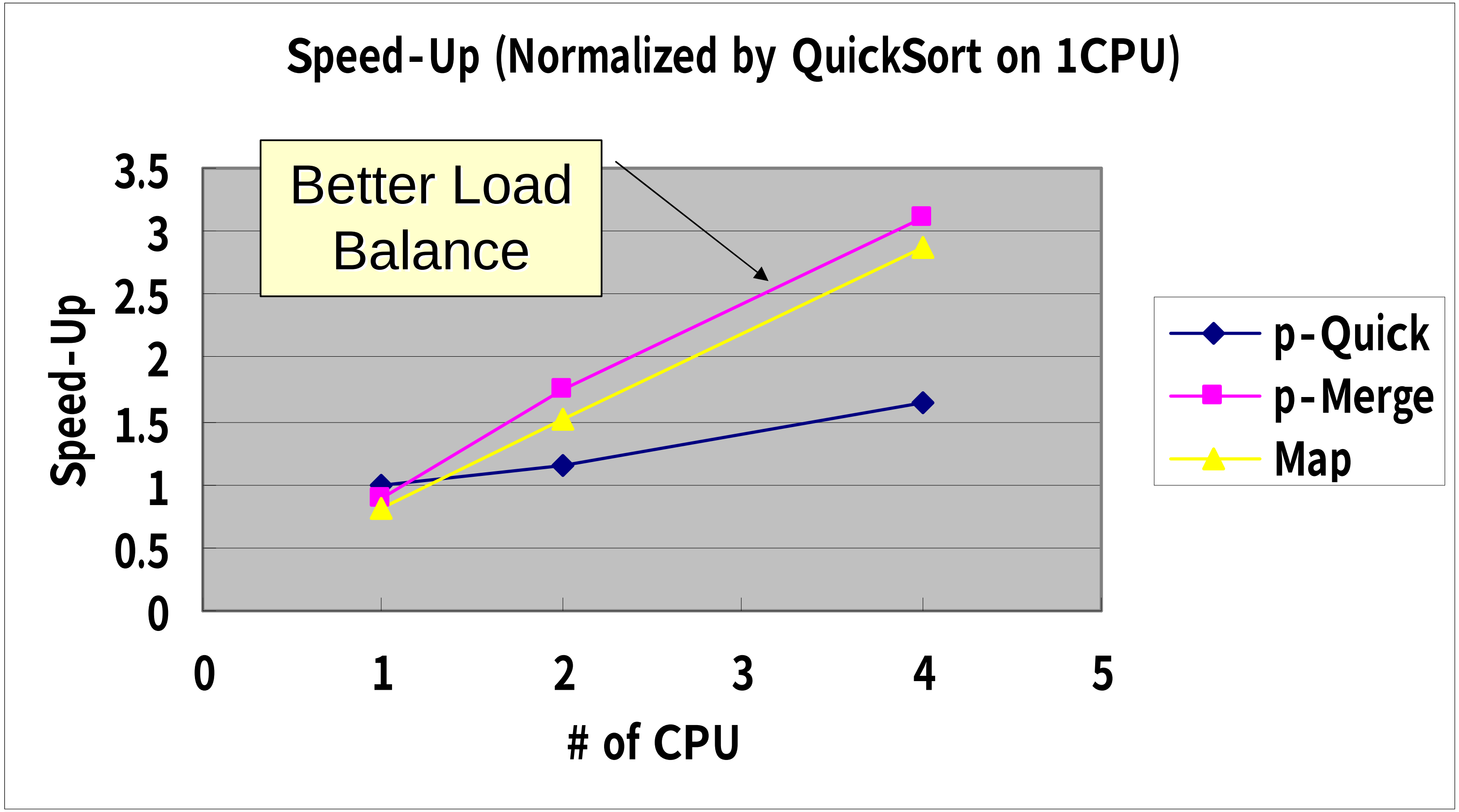
- Good Load Balance
- More Copies

- Good Load Balance
- Less Copies

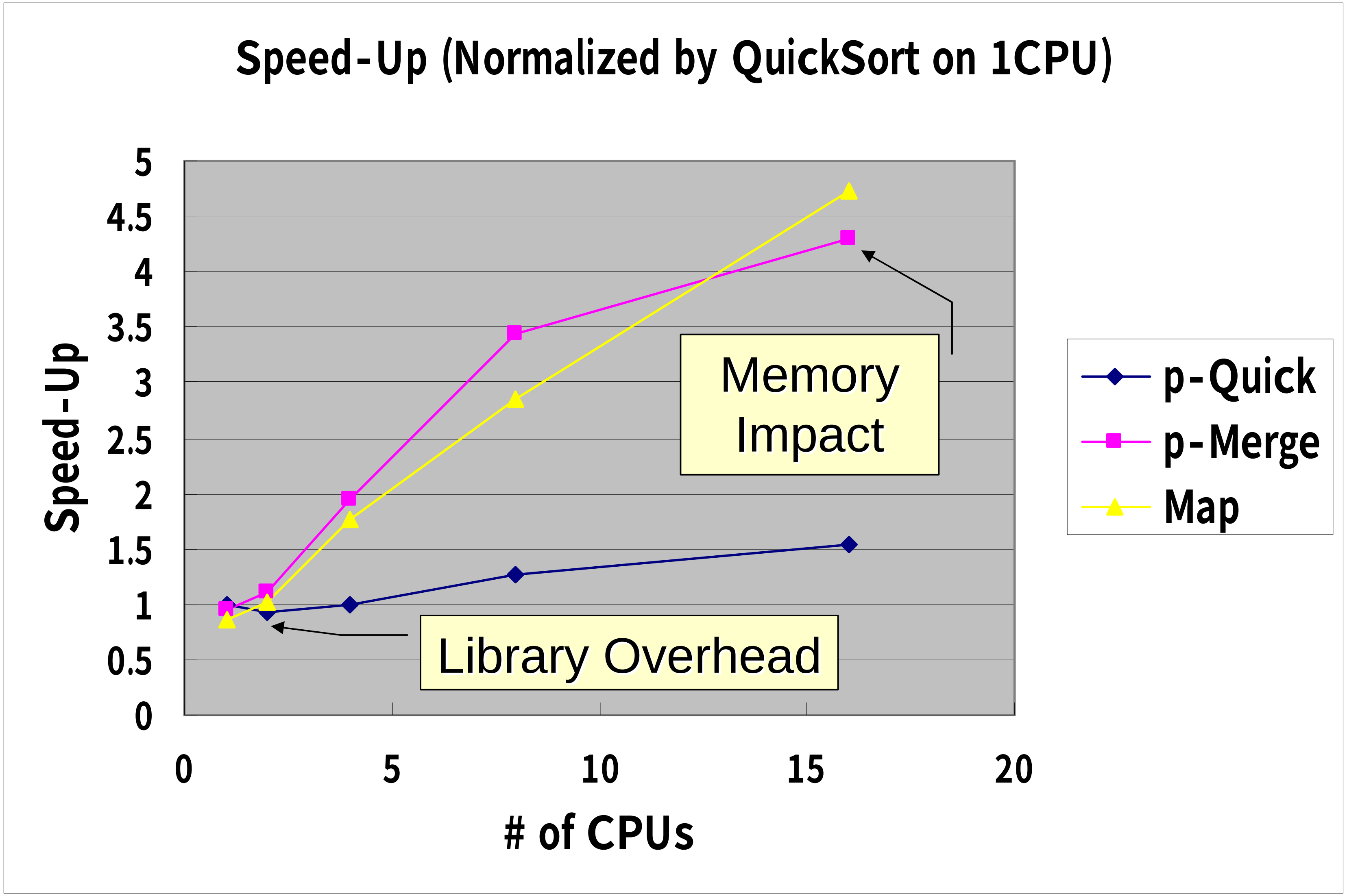
Outline

- Multi-core, Everywhere
  - Scalable Algorithms
  - How to get Performance?
- Example of Scalable Algorithms: Sorting
- Experiments
  - Execute Three Sorting Algorithms on Intel Quad-Core Processor(s)
  - 10 Randomly-Generated Data ($N=10^7$)
- Future Directions

Execution Time (4CPU = Core2 Quad)

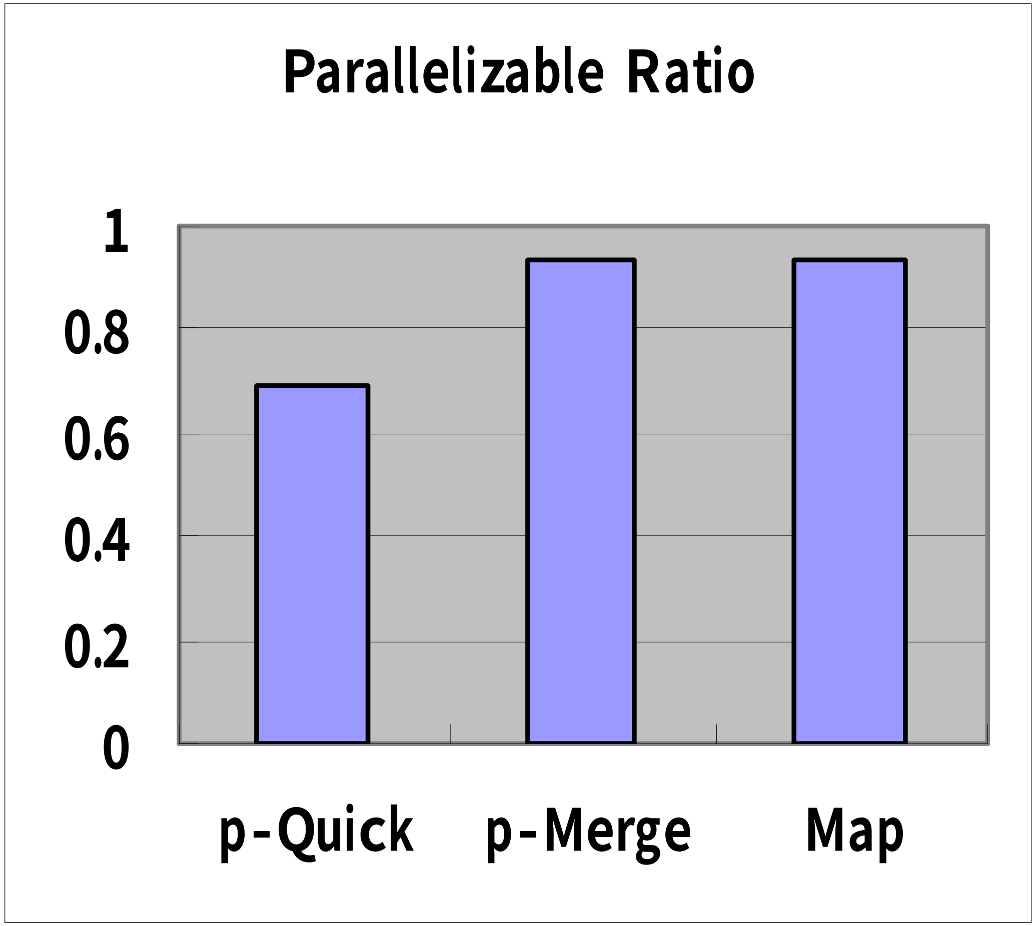


Execution Time (16CPU = Xeon Quad-Core x 4-way)



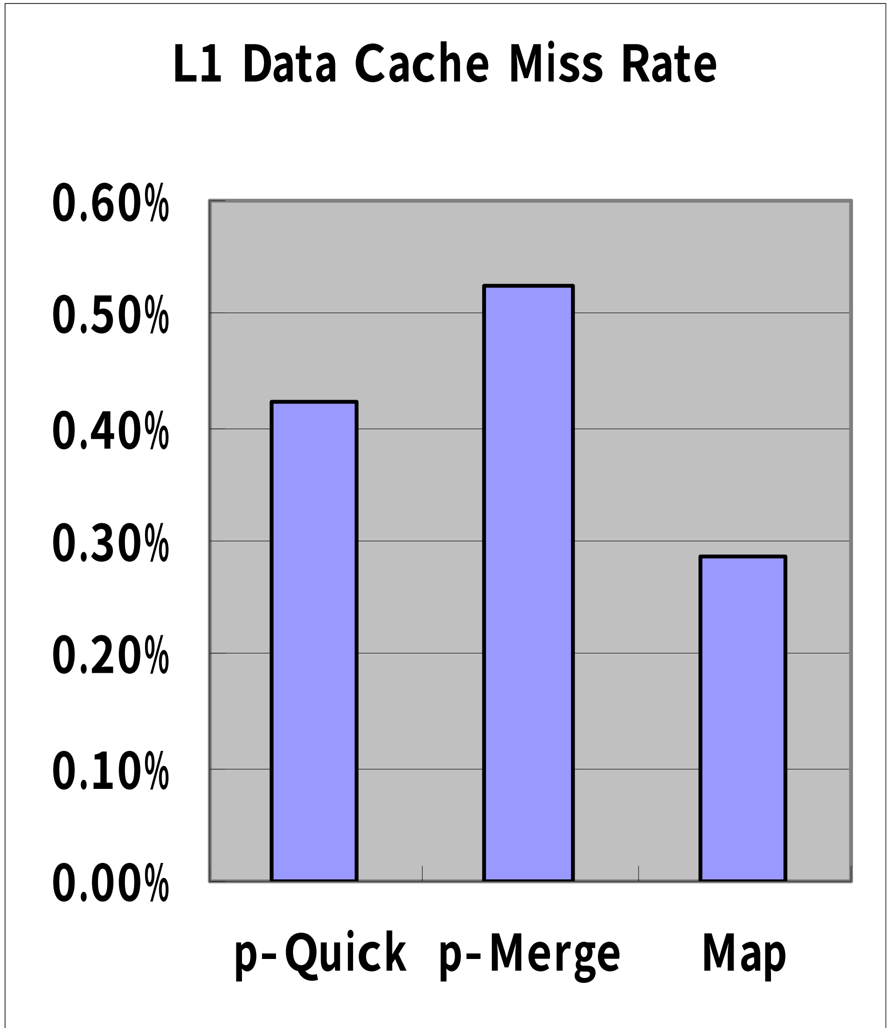
Amdahl's Law

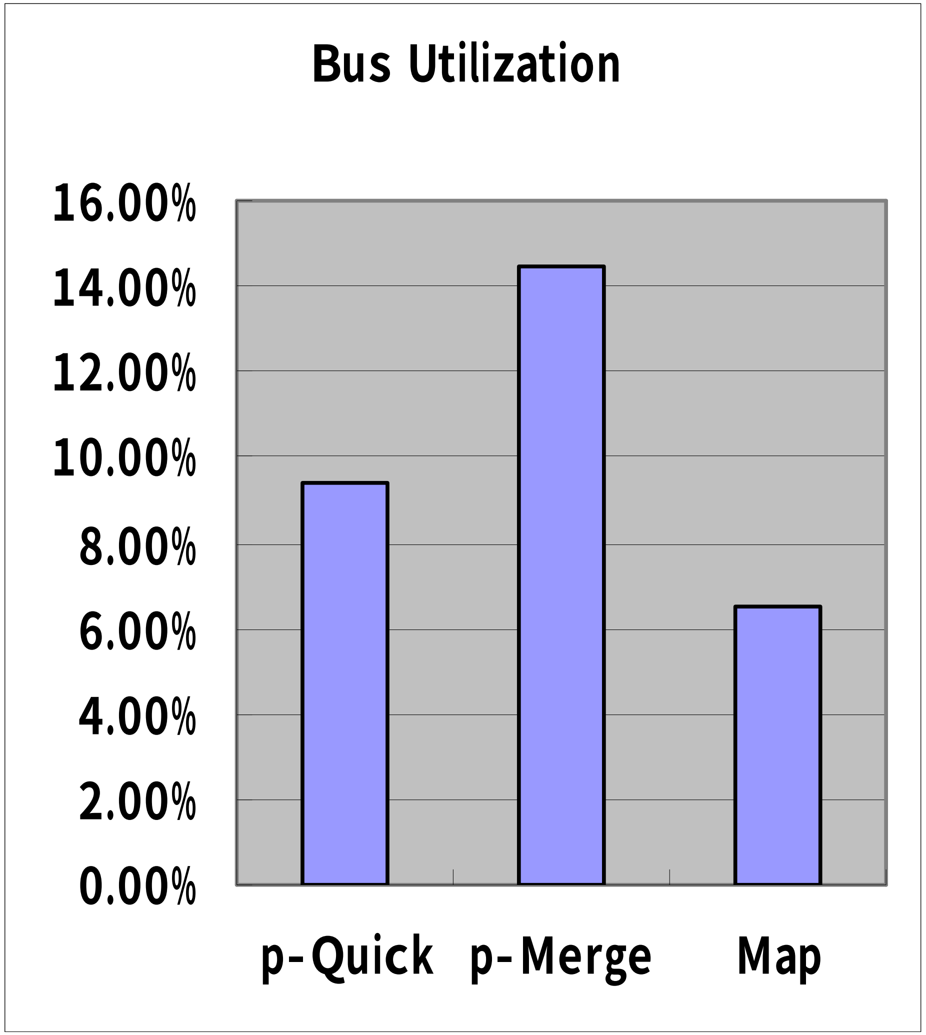
More than 93% of Executed Codes are Parallelizable for Parallel Merge Sort and Map Sort



Memory Access (Data Cache Miss Rate)

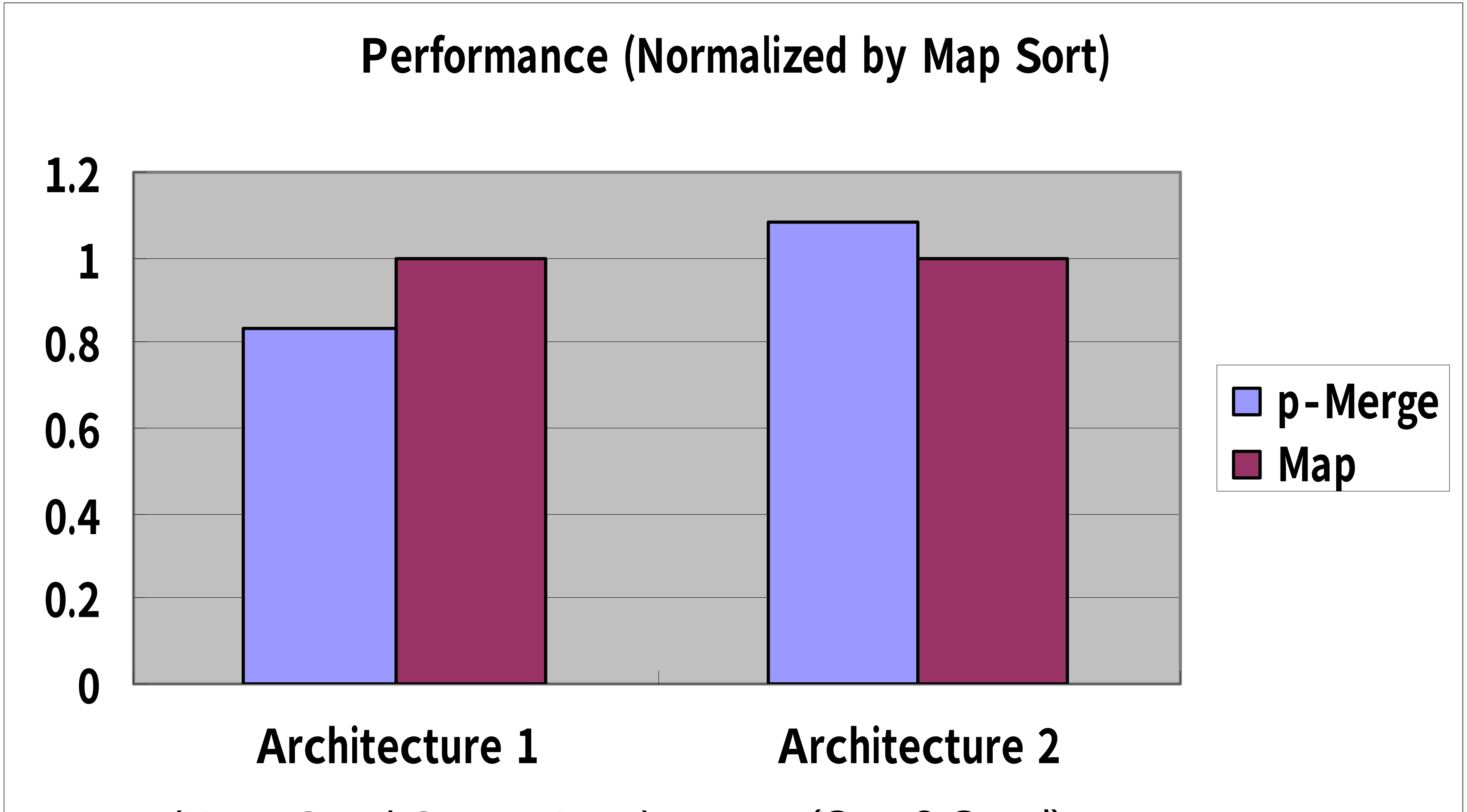
Merge Sort Has More Memory Impact





Comparison on Different Architecture

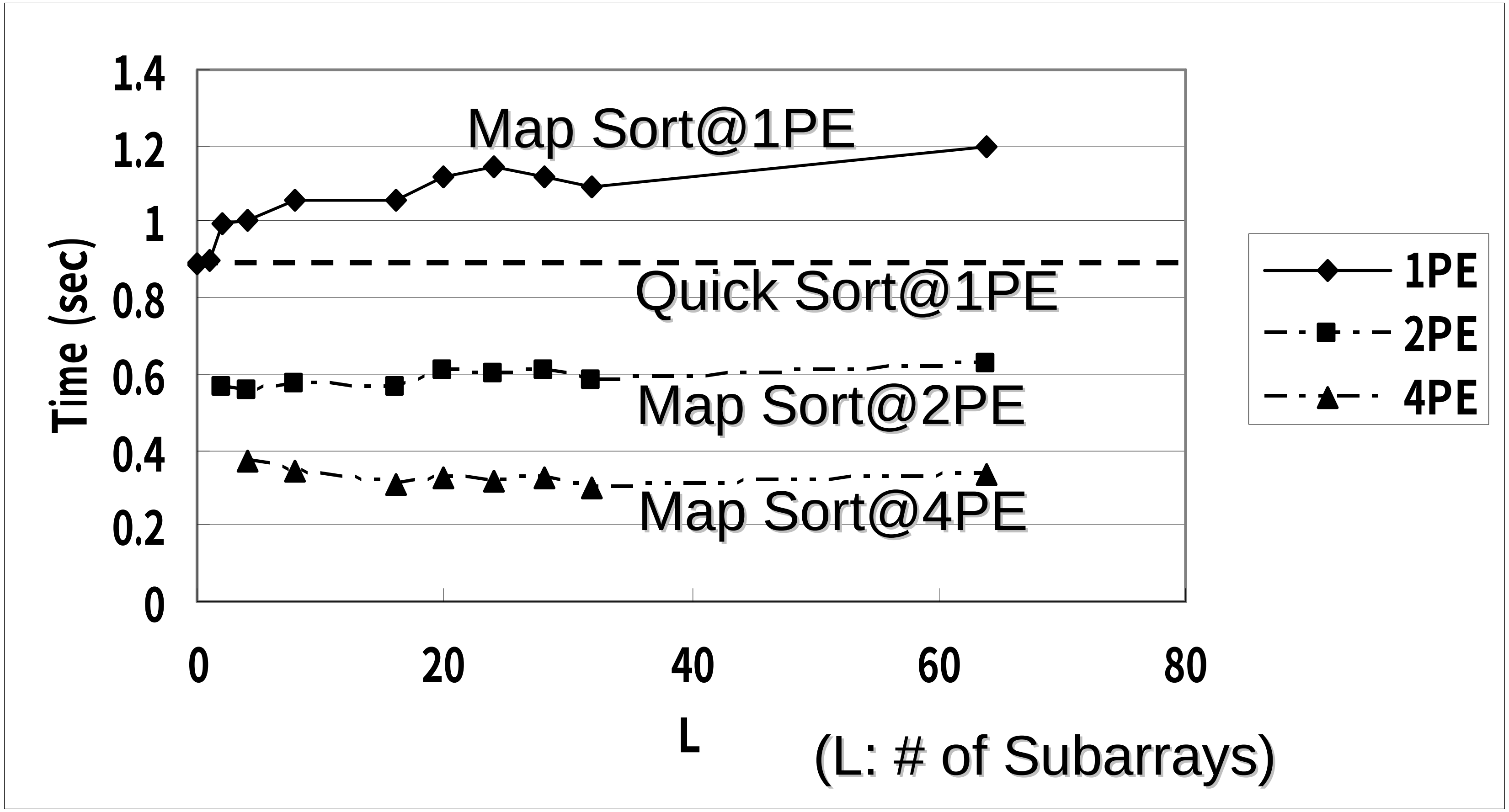
Difference may be Caused by Memory Impact.



(Xeon Quad Core x 4way) (Core2 Quad)

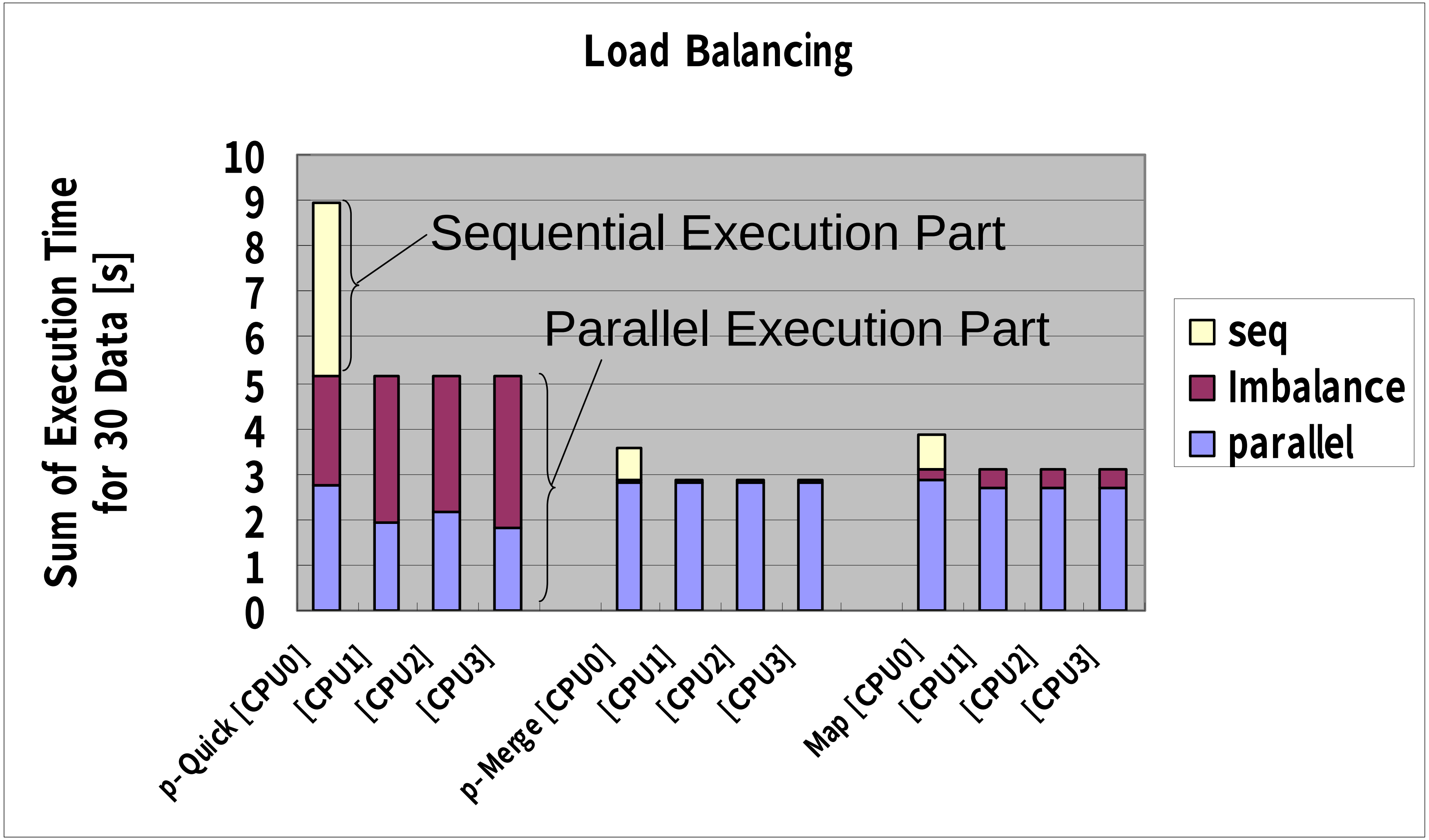
Granularity (Example of Map Sort)

Trade-Off between Performance on 1PE & Speed-Up on Multiple PE



Load Balancing

Parallel Quick Sort Has Worse Load Balancing



Future Directions

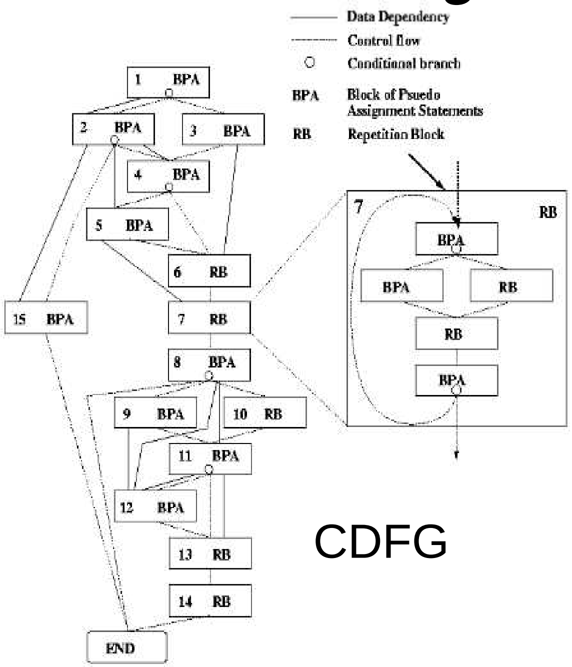
- Even If Algorithms are Scalable,
  We Need to Consider Many Architecture Parameters to Achieve Better Performance.

- Who likes it in EDA Area?

- What can we do?

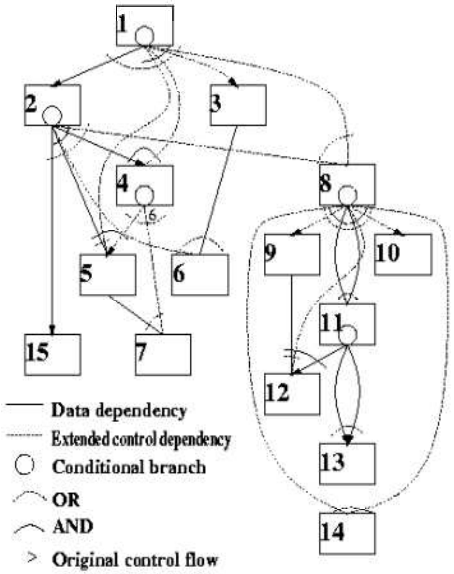
Automatic Parallelizable Compiler

- OSCAR Compiler (Kasahara Lab., Waseda Univ.)
  - Partition Program Codes into Blocks, and Generate Control-Data Flow Graph (CDFG)
  - Analyze CDFG, Consider Architectural Parameters, Parallelize Program Codes for Optimal Execution with Multiple Granularity
  - Thread Assignment Optimization (to CPUs) with Data Placement Optimization on Memory
  - Thread Assignment Optimization for Power Optimization



CDFG

Multi-Grain Parallelization

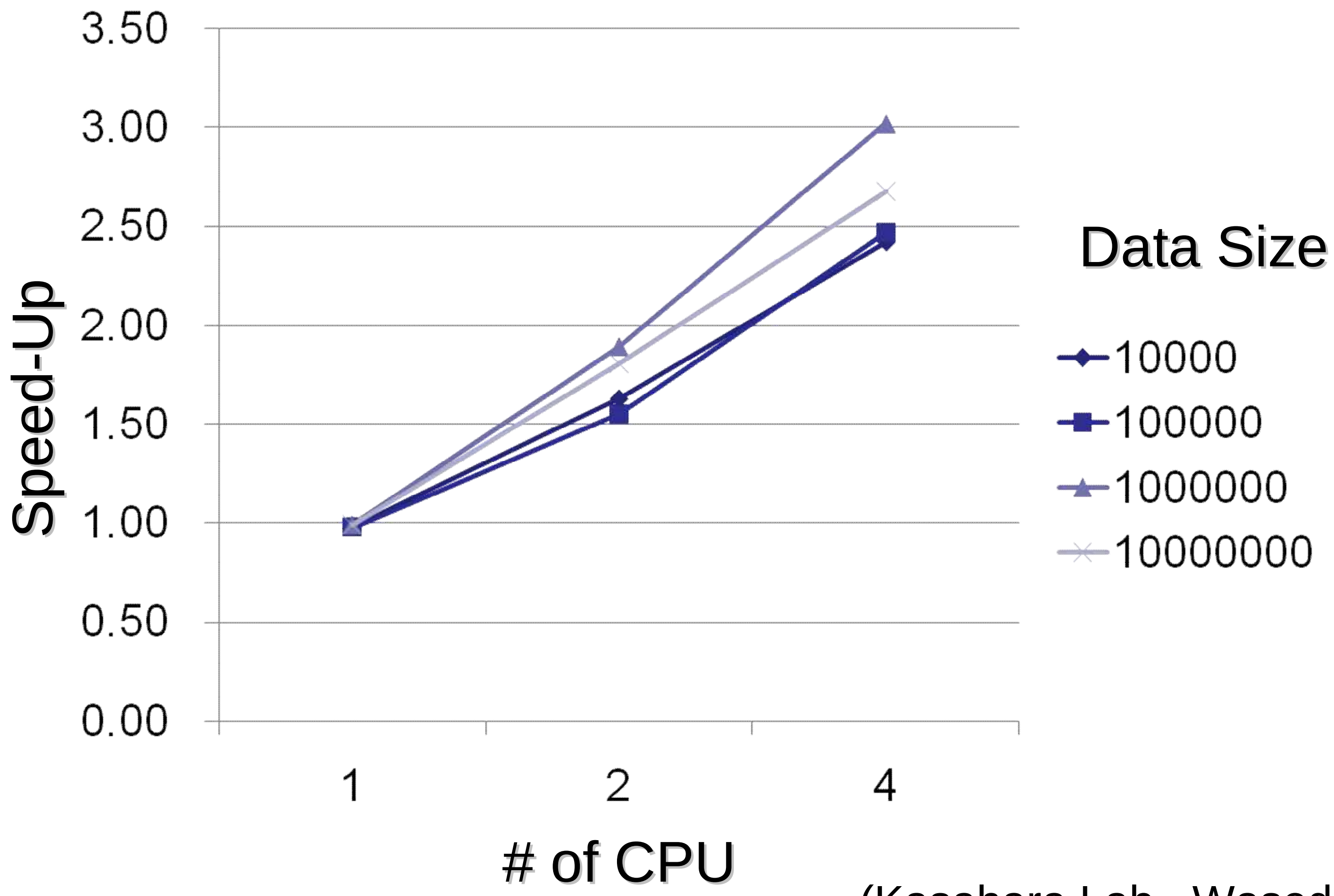


8 is after 7 in CDFG, but no Data dependency between 7 and 8. Then, After Branch (1 to 3, or 2 to 4), 8 is executable.

Reference: http://www.kasahara.cs.waseda.ac.jp/

MapSort

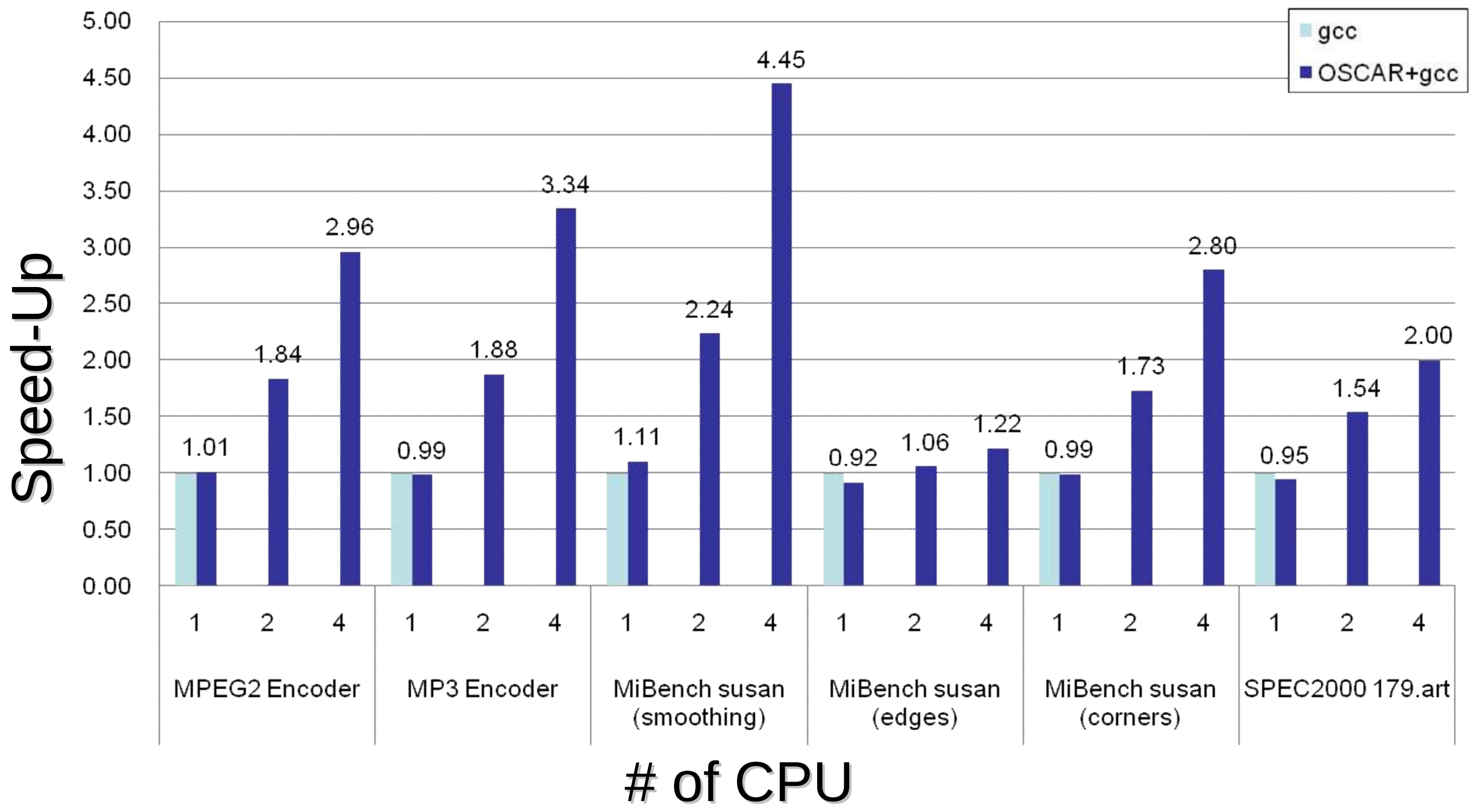
OSCAR Compiler, CPU: MPCore (ARM-NEC)



(Kasahara Lab., Waseda Univ.)

Comparison of OSCAR Compiler with gcc

- (CPU:MPCore)



(Kasahara Lab., Waseda Univ.)

Parallelizable C

(Kasahara Lab., Waseda Univ.)

- **Some Codes are difficult to Analyze by Compilers.**
- **Need Restriction in Syntaxes to Get Better Performance**
  - **Prohibit Recursive Call**
    - **Exclude Recursion from Optimization**
  - **Pointers can be used only in Subscripts of Function Call.**
    - **Don't alias pointer variables**
    - **Don't update pointer variables**

Example

```
int main() {
    int a[30][20][10], b[20][10], c;
    int i;
    …
    input(b);
    for (i=0; i<30; i++) {
        process(a[i], b, &c);
    }
    …
}
```

Caller Subscript
a[20][10], b[20][10], c

```
void process(int a[][10], int b[][1    int *c){
    int t[20];  int i, j;
    …
    for (i = 0; i < 20; i++)
        for  (j = 0; j < 10; j++)
            a[i][j] = b[i][j] * t[i];
    …
    func(a, b[0], t);
    *c += …;
}
```

Don't Alias a, b, c

Conclusions & Future Directions (1)

- **Scalable Algorithms are Indispensable for EDA software on Future Multi-Core-based Computer Systems**
  1. ***P-times Smaller Time Complexity with P CPUs***
  2. ***Comparable Processing Time on a CPU* compared with optimized algorithms for single-CPU processors**
  3. ***Higher Speed-Up with Multiple CPUs***
- **Achieving Scalability is Difficult for Basic Algorithms, Graph & Network Algorithms**
  – **Highly Optimized for Single CPU**

Conclusions & Future Directions (2)

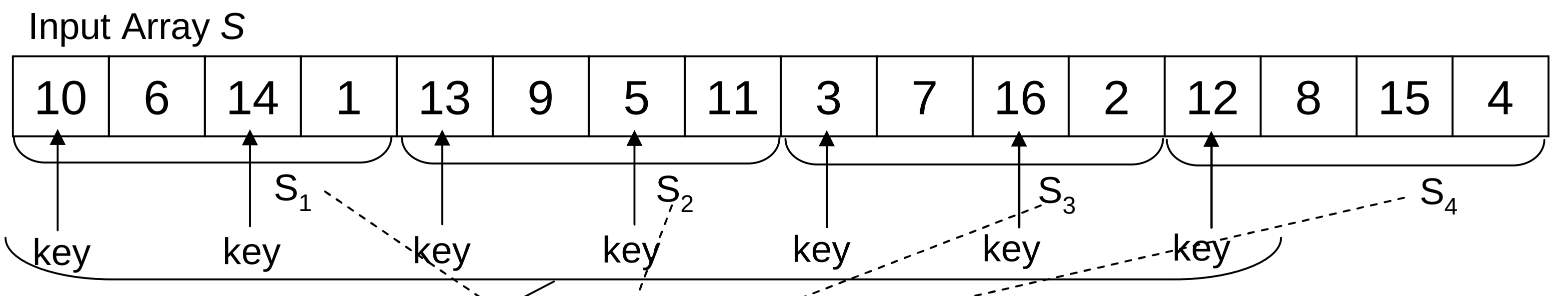
- Sorting is a Typical Example
  - Parallel Quick Sort, Parallel Merge Sort, Map Sort
- Even if Algorithms are Scalable, it is not straightforward to Achieve Performance Scalability on Multi-Cores
- To Achieve Scalability (as Future Directions)
  1. Scalable Algorithms (especially, for Fundamental, Graph & Network Algorithms)
  2. Write Parallelizable C Codes
     - e.g. Parallelizable C with OpenMP
  3. Automatic Parallelizable Compilers

CPU Information

- **Q9550 Core2 Quad@2.83GHz**
  - **L1 64KB per Core**
  - **L2 8MB**

- **X7350 Xeon Quad@2.93GHz**
  - **L1 N/A**
  - **L2 12MB**

Map Sort - - - Example (1)



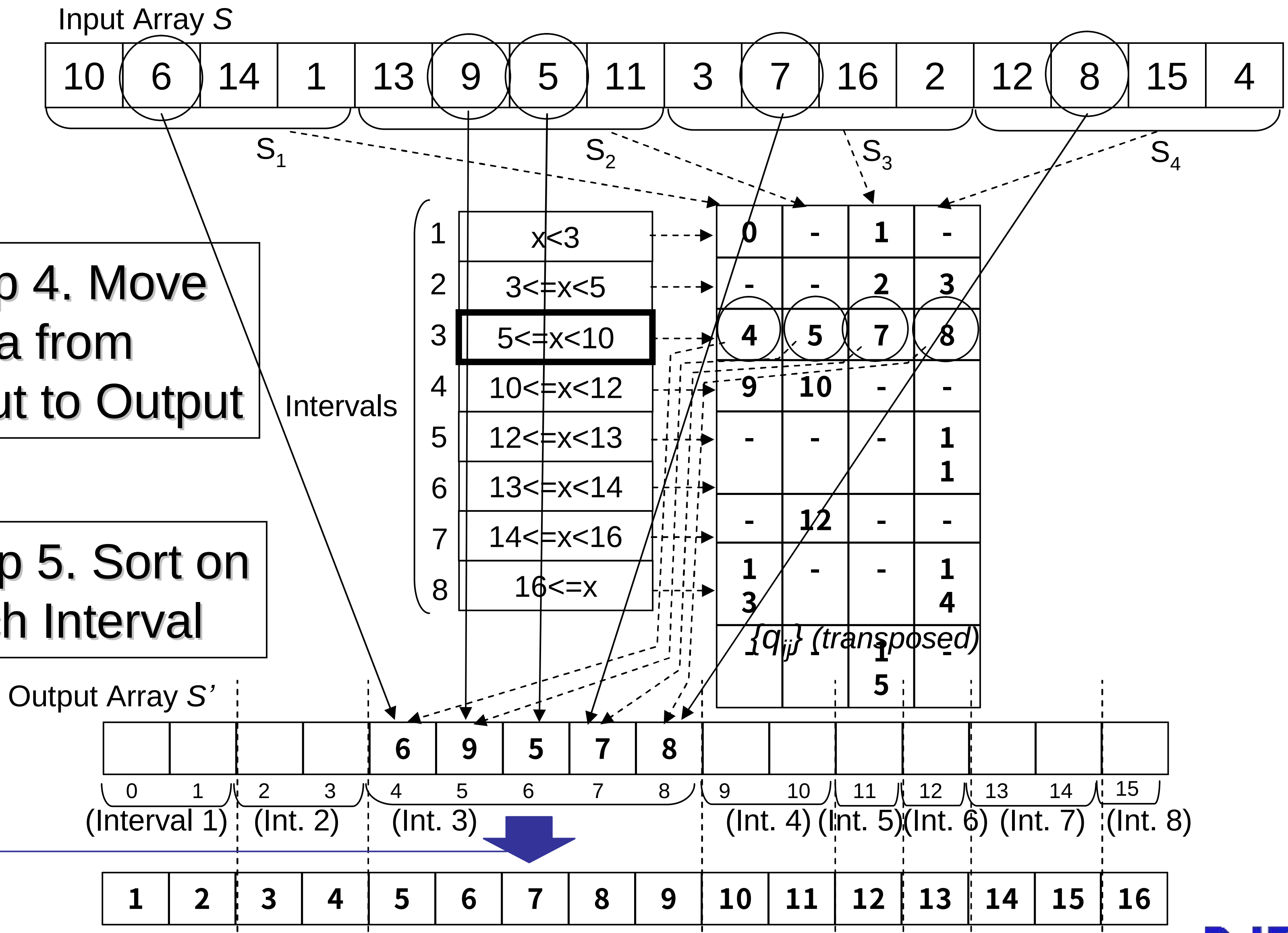
	Intervals						Map			
1	x<3	1	0	1	0		0	-	1	-
2	3<=x<5	0	0	1	1		-	-	2	3
3	5<=x<10	1	2	1	1		4	5	7	8
4	10<=x<12	1	1	0	0		9	10	-	-
5	12<=x<13	0	0	0	1		-	-	-	11
6	13<=x<14	0	1	0	0		-	12	-	-
7	14<=x<16	1	0	0	1		13	-	-	14
8	16<=x	0	0	1	0		-	-	15	-

Step 1. Find Keys,
= Define Intervals

Step 2. Count Data,
each Subset, Intervals

Step 3.
Construct Map

Map Sort - - - Example (2)

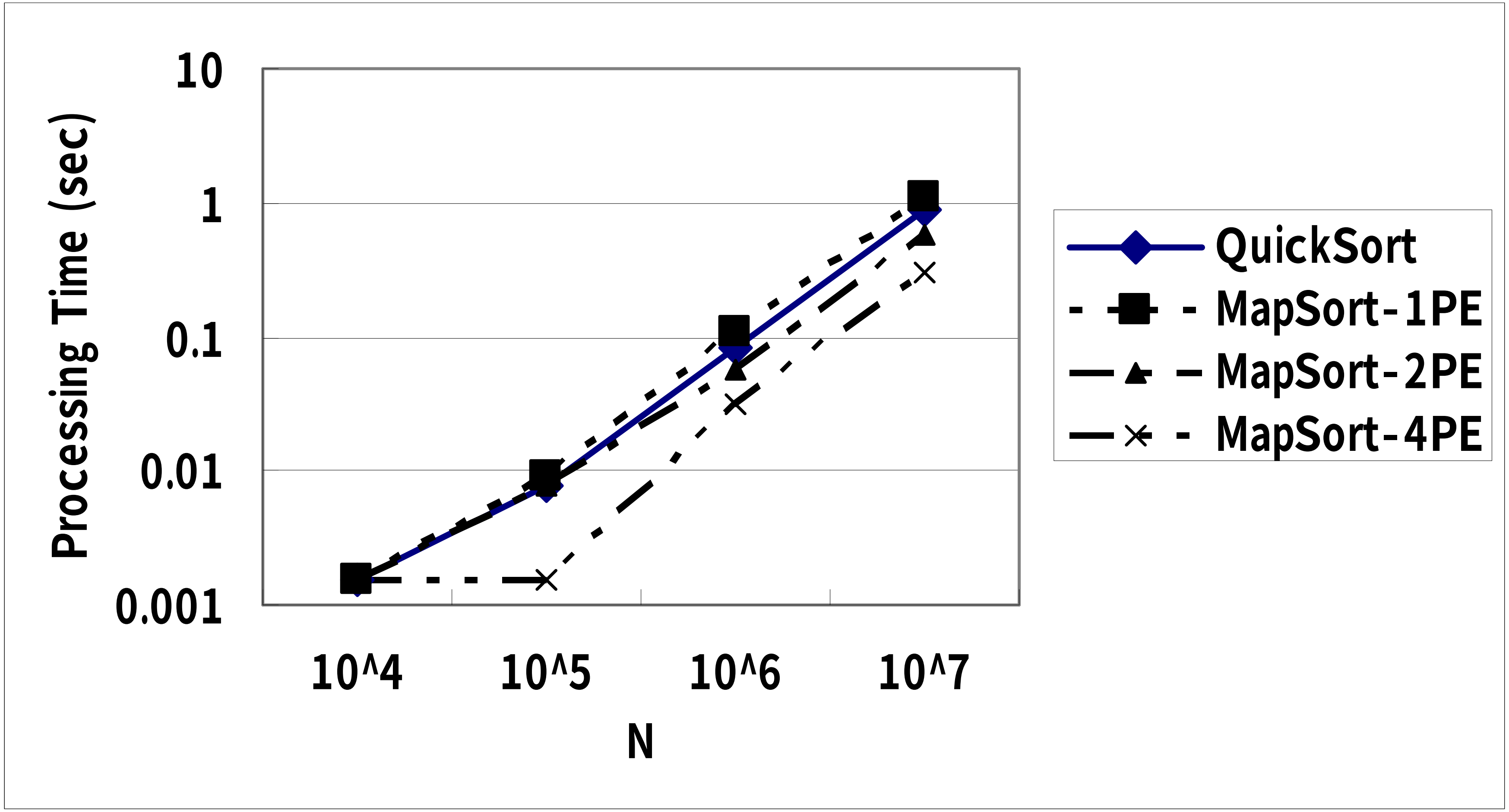


Step 4. Move Data from Input to Output

Step 5. Sort on each Interval

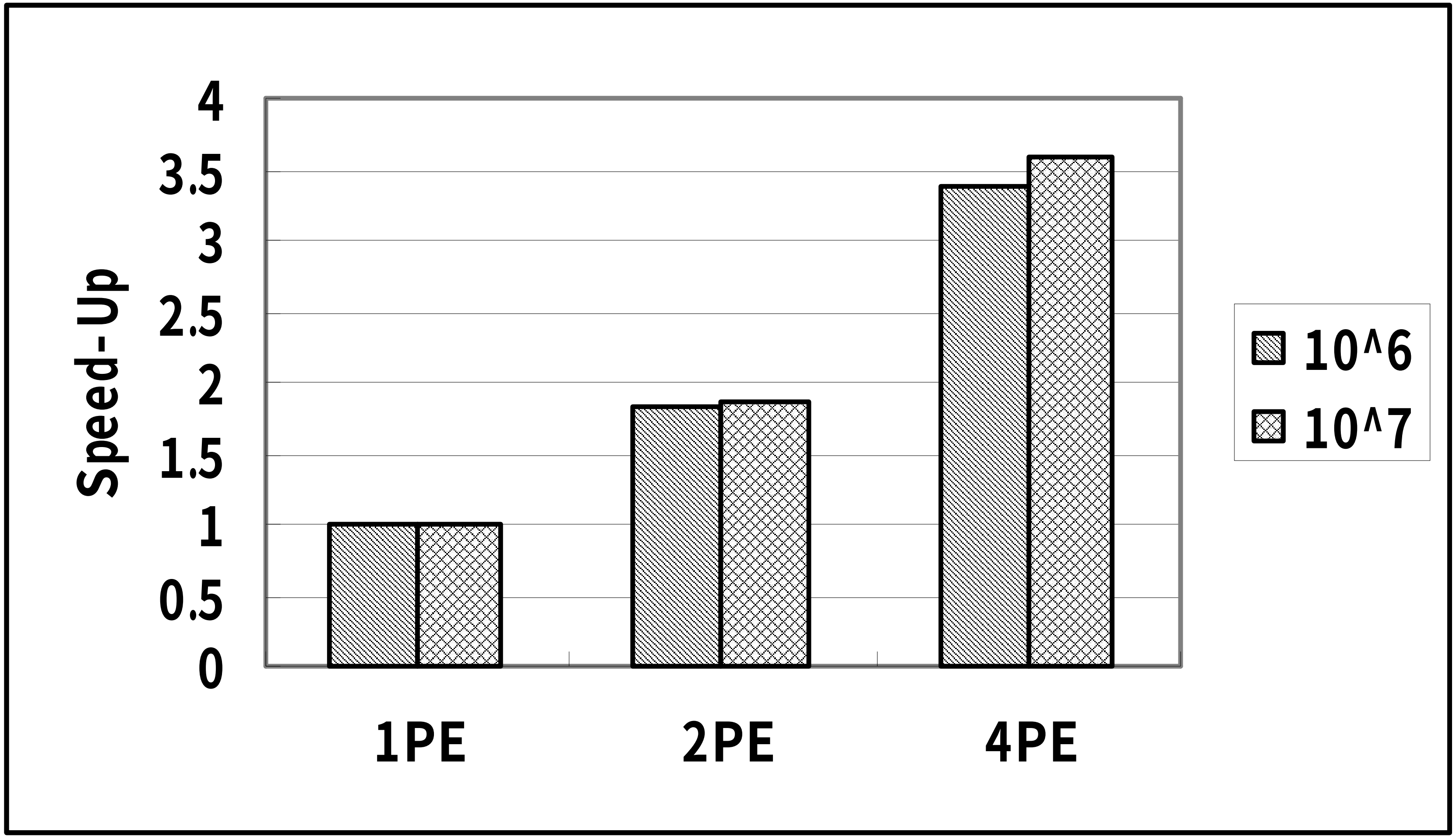
Map Sort --- Experimental Results (1)

Processing Time (Intel Quad Core QX6700, 2.66GHz, OpenMP C)



Map Sort - - - Experimental Results (2)

Speed Up for Map Sort



Map Sort - - - Experimental Results (3)

Comparison with Parallel Quick Sort