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

Masato Edahiro
NEC Corporation
University of Tokyo
• M ulti-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 Initialization 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
Today’s Topic

- Algorithms SHOULD BE SCALABLE
- Even if an Algorithm is Scalable, in Many Cases, its Implementation is NOT Scalable in Performance
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

- Scientific Calculation (for Servers)
- Media Processing (for SIM DMachines, e.g. GPU)

- EDA Acceleration
  - Easy to Parallelize: Meta Heuristics
    - Simulated Annealing, Genetic Algorithms, Neural Networks, 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
• Amdahl's Law
• Memory Access
• Inter-Core Communication
• Granularity
• Load Balancing
Amdahl's Law

<table>
<thead>
<tr>
<th>Parallelizable</th>
<th>100%</th>
<th>80%</th>
<th>50%</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 CPU</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 CPU</td>
<td>4</td>
<td>2.5</td>
<td>1.6</td>
</tr>
<tr>
<td>∞ CPU</td>
<td></td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

Faster and Faster  \( \leq 2 \) At Most  \( \leq 2 \)
Memory Access & Inter-Core Communication

- Memory Access and Inter-Core Communication (using Shared Memory) is often critical for Performance
  - Multi-Core Processor
    - MEM Cache Miss, Cache Miss, Bus Congestion, Memory Latency
  - Single Processor
    - Coherent Cache

Diagram showing single processor and multi-core processor architectures with emphasis on cache coherence and memory management.
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

Granularity = 4 × 100 Cycle

1 CPU

- Slower than Before

Granularity=4 × 10000 Cycles

4 CPU

- 1000 Cycles

- □ 3.3
Load Balancing

- Fine Grain: Better Load Balancing
- Coarse Grain: Smaller Overhead
• 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

- 1CPU
- 2CPU
- 4CPU

### Parallel Merge Sort

- Merge w/ \((P/2)\)CPUs
- Merge w/ \((P/2)\)CPUs
- Merge w/ \(P\) CPUs
- Not Easy to Utilize Multi-Core in First Several Steps
- Often Makes Load Imbalance
Parallel Merge Sort (1)

4CPU

Sort

Merge

Merge

2CPU x 2

Sort

Merge

Sort

• Very Good Load Balancing
• Merge with 2 CPUs
• 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 2 CPUs

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

Input Data

Recursive Partitioning
(Sequential at first)

Int. Int. Int. ... Int.
(Ordered Intervals)

Sorting
(Interval-Level Parallelism)

Sorted Data

Parallel Merge Sort

Input Data

$\text{Subset} \rightarrow \text{Subset} \rightarrow \text{Subset} ... \rightarrow \text{Subset}$
(Subset-Level Parallelism)

Sorting

Recursive Merge
(Merge w/ 2k-CPUs)

Sorted Data

Map Sort
[Edahiro 2007]

Input Data

$\text{Subset} \rightarrow \text{Subset} \rightarrow \text{Subset} ... \rightarrow \text{Subset}$
(Subset-Level Parallelism)

Mapping
(Parallelizable)

Int. Int. Int. ... Int.
(Ordered Intervals)

Sorted Data

- Sequential Part
- Load Imbalance
- Good Load Balance
- More Copies
Step 1. Find Keys, = Define Intervals

Input Data

Output Data

Intervals

3 18 7

x<3 3<=x<7 7<=x<18 18<=x
Input Data

Step 2. Count Data, each Subset, Intervals

Step 3. Construct Map

Output Data

Subsets

Intervals

Map (Subsets to Intervals)
Step 4. Move Data from Input to Output

ONLY DATA COPY

Step 5. Sort on each Interval

Output Data

Input Data

Map

Subsets

PE1

PE2

PE3

PE4

Intervals

SORT
PE1

SORT
PE2

SORT
PE3

SORT
PE4
Map Sort --- Parameters

- **Input Data**
  - M Subsets

- **Output Data**
  - L Intervals

- **Map**

N Data

- **Data**

3 7 18
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: $O((N/P) \log (N/P)) = O((N/P) \log N)$
Parallel Quick Sort

- Input Data
- Recursive Partitioning (Sequential at first)
- Int. Int. Int. ... Int. (Ordered Intervals)
- Sorting (Interval-Level Parallelism)
- Sorted Data

- • Sequential Part
- • Load Imbalance

Parallel Merge Sort

- Input Data
- Subset Subset Subset ... Subset (Subsets)
- Sorting (Subset-Level Parallelism)
- Recursive Merge (Merge w/ 2k-CPUs)
- Sorted Data

- • Good Load Balance
- • More Copies

Map Sort [Edahiro 2007]

- Input Data
- Subset Subset Subset ... Subset (Subsets)
- Mapping (Parallelizable)
- Sorting (Interval-Level Parallelism)
- Sorted Data

- • Good Load Balance
- • Less Copies
• 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)

Speed-Up (Normalized by QuickSort on 1CPU)

0 0.5 1 1.5 2 2.5 3 3.5

# of CPU

Better Load Balance

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

Speed-Up (Normalized by QuickSort on 1CPU)

0 0.5 1 1.5 2 2.5 3 3.5 4 4.5 5

0 5 10 15 20

# of CPUs

Speed-Up

Map

Memory Impact

Library Overhead

© NEC Corporation 2009
More than 93% of Executed Codes are Parallelizable for Parallel Merge Sort and Map Sort
Merge Sort Has More Memory Impact

[Bar chart showing memory access data for different algorithms]
Difference may be Caused by Memory Impact.
Trade-Off between Performance on 1PE & Speed-Up on Multiple PE

(L: # of Subarrays)
Parallel Quick Sort Has Worse Load Balancing

Sequential Execution Part

Parallel Execution Part

p-Quick [CPU0] [CPU1] [CPU2] [CPU3]
p-Merge [CPU0] [CPU1] [CPU2] [CPU3]
Map [CPU0] [CPU1] [CPU2] [CPU3]

Sum of Execution Time

for 30 Data [s]

© NEC Corporation 2009
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

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

Multi-Grain Parallelization

- OSCAR Compiler (KasaharaLab., WasedaUniv.)
  - 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

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/
OSCAR Compiler, CPU: MPCore (ARM-NEC)

(Kasahara Lab., Waseda Univ.)
### Comparison of OSCAR Compiler with gcc

<table>
<thead>
<tr>
<th>Application</th>
<th># of CPU</th>
<th>gcc Speed-Up</th>
<th>OSCAR+gcc Speed-Up</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPEG2 Encoder</td>
<td>1</td>
<td>1.01</td>
<td>2.00</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.84</td>
<td>2.96</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>0.99</td>
<td>3.34</td>
</tr>
<tr>
<td>MP3 Encoder</td>
<td>1</td>
<td>1.11</td>
<td>2.24</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.88</td>
<td>3.34</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>0.92</td>
<td>4.45</td>
</tr>
<tr>
<td>MiBench susan</td>
<td>1</td>
<td>0.92</td>
<td>1.06</td>
</tr>
<tr>
<td>(smoothing)</td>
<td>2</td>
<td>1.06</td>
<td>1.22</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>0.99</td>
<td>1.73</td>
</tr>
<tr>
<td>MiBench susan</td>
<td>1</td>
<td>0.95</td>
<td>1.54</td>
</tr>
<tr>
<td>(edges)</td>
<td>2</td>
<td>1.54</td>
<td>2.00</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>0.95</td>
<td>2.00</td>
</tr>
<tr>
<td>MiBench susan</td>
<td>1</td>
<td>0.95</td>
<td>1.54</td>
</tr>
<tr>
<td>(corners)</td>
<td>2</td>
<td>1.54</td>
<td>2.00</td>
</tr>
<tr>
<td>SPEC2000 179.art</td>
<td>1</td>
<td>0.95</td>
<td>1.54</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.54</td>
<td>2.00</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>0.95</td>
<td>2.00</td>
</tr>
</tbody>
</table>

(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

```c
void process(int a[][10], int b[][10], 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 += ...;
}
```

```c
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); 
  ... 
}
```

Example

(Kasahara Lab., Waseda Univ.)
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@2.83G Hz
    – L1 64KB per Core
    – L2 8MB
  • X7350 Xeon Quad@2.93G Hz
    – L1 N/A
    – L2 12MB
Input Array $S$

<table>
<thead>
<tr>
<th></th>
<th>10</th>
<th>6</th>
<th>14</th>
<th>1</th>
<th>13</th>
<th>9</th>
<th>5</th>
<th>11</th>
<th>3</th>
<th>7</th>
<th>16</th>
<th>2</th>
<th>12</th>
<th>8</th>
<th>15</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>key</td>
<td></td>
<td>S_1</td>
<td></td>
<td>key</td>
<td>S_2</td>
<td></td>
<td>key</td>
<td>S_3</td>
<td></td>
<td>key</td>
<td>S_4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Intervals

- $x < 3$
- $3 \leq x < 5$
- $5 \leq x < 10$
- $10 \leq x < 12$
- $12 \leq x < 13$
- $13 \leq x < 14$
- $14 \leq x < 16$
- $16 \leq x$

Step 1. Find Keys, = Define Intervals
Step 2. Count Data, each Subset, Intervals
Step 3. Construct Map
Step 4. Move Data from Input to Output

Step 5. Sort on each Interval
Processing Time (Intel Quad Core QX6700, 2.66GHz, OpenMP C)
Speed Up for Map Sort
Comparison with Parallel Quick Sort

- Speed Up (Normalized by Quicksort@1PE)
- # of PE
- QuickSort
- MapSort