A Case for Parallelism Profilers and Advisers with What-If Analyses

Santosh Nagarakatte
Rutgers University, USA

@ Workshop on Dependable and Secure Software Systems @ ETH Zurich, October 2019
Is Parallel Programming Hard, And, If So, What Can You Do About It?

“Parallel programming has earned a reputation as one of the most difficult areas a hacker can tackle. Papers and textbooks warn of the perils of deadlock, livelock, race conditions, non-determinism, Amdahl’s-Law limits to scaling, and excessive realtime latencies. And these perils are quite real; we authors have accumulated uncounted years of experience dealing with them, and all of the emotional scars, grey hairs, and hair loss that go with such experiences.”

[McKenny:arXiv17]

Main reasons: use of the wrong abstraction, lack of performance analysis and debugging tools
Illustrative Example

Write a parallel program

Given a range of integers (0 to n)

Find all the prime numbers in the range

Perform a computation on the primes

Output result

Student in my class
for(int i=0; i<n; ++i) compute(i);
Illustrative Example

Divide the range into 4 parts and perform computation

Identify the number of processors on the machine (4)

Run: ./primes

Speedup: 1.8X over serial execution

Load Imbalance

Why?
Need to write Performance Portable Code - Advocacy for Task Parallelism

Express all the parallelism as tasks

Runtime that dynamically balances load by assigning tasks to idle threads

1 2 3 4 .......... n

T_1 T_2 T_3 .......... T_m

P_1 P_2 .......... P_k

Rutgers
Illustrative Example

Expresses parallel work in terms of tasks

Run: ./primes_tasks

Speedup: 3.8X over serial execution on 4 cores

Is it performance portable?

Student in my class
Performance Debugging Tools

- Most of them provide info on frequently executed regions.
- Critical path information is useful.
- Coz [SOSP 2015]: Identifies if a line of code matters in increasing speedup on a given machine.
Our Parallelism Profilers and Advisers:
TaskProf & OMP-WHIP [FSE 2017, SC 2018, PLDI 2019]

• Making a case for measuring logical parallelism
  Series-parallel relations + fine-grained measurements is a performance model

• Where should programmer focus?
  Regions with low parallelism => serialization. Critical path!

• Does it matter?
  Automatically identify regions to increase parallelism to a threshold

What-if Analyses - mimic the effect of parallelization
Differential analyses to identify regions with secondary effects
General for multiple parallelism models. This talk focuses on OpenMP
Performance Model for Logical Parallelism and What-If Analyses
Performance Model for Computing Parallelism

- Profile on a machine with low core count and identify scalability bottlenecks
- OSPG: Logical series-parallel relations between parts of a OpenMP program
  - Inspired by prior work: DPST [PLDI 2012], SP Parse tree [SPAA 2015]
OpenMP Series Parallel Graph (OSPG)

• A data structure to capture series-parallel relations
  • Inspired by Dynamic Program Structure Tree [PLDI 2012]
  • OSPG is an ordered tree in the absence of task dependencies in OpenMP

• Handles the combination of work-sharing (fork-join programs with threads) and tasking

• Precisely captures the semantics of OpenMP
  • Three kinds of nodes: W, S, and P nodes similar to Async, Finish, and Step nodes in the DPST
Code Fragments in OpenMP Programs

A code fragment is the longest sequence of instructions in the dynamic execution before encountering an OpenMP construct.
Capturing Series-Parallel Relation with the OSPG

**W-nodes** capture computation
A maximal sequence of dynamic instructions between two OpenMP directives

**P-nodes** capture the parallel relation
Nodes in the sub-tree of a P-node logically executes in parallel with right siblings of the P-node

**S-nodes** capture the series relation
Nodes in the sub-tree of a S-node logically executes in series with right siblings of the S-node
Determine the series-parallel relation between any pair of nodes with an LCA query.

Check the type of the LCA’s child on the path to the left w-node. If it’s a p-node, they execute in parallel. Otherwise, they execute in series.

S2 = LCA(W2,W3)
P1 = Left-Child(S2,W2,W3)
Capturing Series-Parallel Relation with the OSPG

Determine the series-parallel relation between any pair of nodes with an LCA query.

Check the type of the LCA's child on the path to the left w-node. If it’s a p-node, they execute in parallel. Otherwise, they execute in series.

- S1 = LCA(W2,W4)
- S2 = Left-Child(S1,W2,W4)
Profiling an OpenMP Merge Sort Program

- Merge sort program parallelized with OpenMP

```c
void main()
    
    #pragma omp parallel
    #pragma omp single
    mergeSort(arr, 0, n);

void mergeSort(int* arr, int s, int e){
    if (n <= CUT_OFF)
        serialSort(arr, s, e);
    int mid = s + (e-s)/2;
    #pragma omp task
    mergeSort(arr, s, mid);
    #pragma omp task
    mergeSort(arr, mid+1, e);
    #pragma omp taskwait
    merge(arr, s, e);
}
```
OSPГ Construction

```c
void main(){
    int* arr = init(&n);
    #pragma omp parallel
    #pragma omp single
    mergeSort(arr, 0, n);
}
```
void mergeSort(int* arr, int s, int e) {
    if (n <= CUT_OFF)
        serialSort(arr, s, e);
    int mid = s + (e-s)/2;
    #pragma omp task
    mergeSort(arr, s, mid);
    #pragma omp task
    mergeSort(arr, mid+1, e);
    #pragma omp taskwait
    merge(arr, s, e);
}
Parallelism Computation Using OSPG
Compute Parallelism

Measure work in each Work node with fine grained measurements

Compute work for each internal node
Compute Serial Work

Measure work in each Work node

Compute work for each internal node

Identify serial work on critical path
Compute Serial Work

Measure work in each Work node

Compute work for each internal node

Compute serial work for each Internal node
Source Code Attribution

Aggregate parallelism at OpenMP constructs
### Parallelism Profile

<table>
<thead>
<tr>
<th>Line Number</th>
<th>Work</th>
<th>Serial Work</th>
<th>Parallelism</th>
<th>Critical Path Work %</th>
</tr>
</thead>
<tbody>
<tr>
<td>program:1</td>
<td>260</td>
<td>160</td>
<td>1.625</td>
<td>3.75</td>
</tr>
<tr>
<td>omp parallel:3</td>
<td>254</td>
<td>154</td>
<td>1.65</td>
<td>33.75</td>
</tr>
<tr>
<td>omp task:11</td>
<td>100</td>
<td>100</td>
<td>1.00</td>
<td>62.5</td>
</tr>
<tr>
<td>omp task:13</td>
<td>100</td>
<td>100</td>
<td>1.00</td>
<td>0</td>
</tr>
</tbody>
</table>
Identify what parts of the code matter in increasing parallelism.
Adviser mode with What-If Analyses

Identify code regions that must be optimized to increase parallelism

Which region to select?

Select a region to optimize

Select step node performing highest work on critical path

Rutgers
Adviser mode with What-If Analyses

Identify code regions that must be optimized to increase parallelism

Select highest step node on critical path

Repeat until threshold parallelism is reached

Identify all W-nodes corresponding to the region and perform what-if analyses

<table>
<thead>
<tr>
<th>Line</th>
<th>Work</th>
<th>Cwork</th>
<th>Parallelism</th>
<th>CP</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>260</td>
<td>85</td>
<td>3.05</td>
<td>7.05%</td>
</tr>
<tr>
<td>3</td>
<td>254</td>
<td>79</td>
<td>1.65</td>
<td>63.5%</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>25</td>
<td>4.00</td>
<td>29.45%</td>
</tr>
<tr>
<td>13</td>
<td>100</td>
<td>25</td>
<td>4.00</td>
<td>0%</td>
</tr>
</tbody>
</table>

W0 S0 S1 P0 W1 S2 W4 P2 P3 W2 W3
Tasking and Scheduling Overhead

Parallelism

Runtime overhead

Speedup

- S0
- S1
- W0
- S2
- W1
- S4
- W4
- P0
- P2
- P3
- W2
- W3

Rutgers
Adviser mode with What-If Analyses

Identify code regions that must be optimized to increase parallelism

Select highest step node on critical path

Repeat until threshold parallelism is reached

OR

Work of highest step node < K * average tasking overhead

What-If Profile

<table>
<thead>
<tr>
<th>Line</th>
<th>Work</th>
<th>Cwork</th>
<th>Parallelism</th>
<th>CP</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>260</td>
<td>85</td>
<td>3.05</td>
<td>7.05%</td>
</tr>
<tr>
<td>3</td>
<td>254</td>
<td>79</td>
<td>1.65</td>
<td>63.5%</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>25</td>
<td>4.00</td>
<td>29.45%</td>
</tr>
<tr>
<td>13</td>
<td>100</td>
<td>25</td>
<td>4.00</td>
<td>0%</td>
</tr>
</tbody>
</table>
Recap

OpenMP program

Performance model

Logical series-parallel relations

+ Work measurements

Parallelism Profile

<table>
<thead>
<tr>
<th>Line</th>
<th>Work</th>
<th>Cwork</th>
<th>Parallelism</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>160</td>
<td>130</td>
<td>1.23</td>
</tr>
<tr>
<td>.....</td>
<td>.....</td>
<td>.....</td>
<td>.....</td>
</tr>
</tbody>
</table>

What-if Regions

<table>
<thead>
<tr>
<th>Region</th>
<th>Parallelization</th>
</tr>
</thead>
<tbody>
<tr>
<td>12-14</td>
<td>4X</td>
</tr>
<tr>
<td>.....</td>
<td>.....</td>
</tr>
</tbody>
</table>

What-if Profile

<table>
<thead>
<tr>
<th>Line</th>
<th>Work</th>
<th>Cwork</th>
<th>Parallelism</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>160</td>
<td>130</td>
<td>16.12</td>
</tr>
<tr>
<td>.....</td>
<td>.....</td>
<td>.....</td>
<td>.....</td>
</tr>
</tbody>
</table>
Differential Analysis to Identify Secondary Effects
Beyond Parallelism - Secondary Effects

• Program can have high parallelism, but low speedup
  • Secondary effects of parallel execution on hardware

• Contention for a system resource
  • Cache – False sharing
  • Memory – High remote memory accesses
  • LLC misses - Reduced locality
  • Processor to data affinity
Differential Analysis

Oracle Performance model

Parallel Execution’s Performance model

Work inflation in region with secondary effects
Inflation over Multiple Metrics

<table>
<thead>
<tr>
<th>Differential Counters</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycles</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HITM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Remote DRAM accesses</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Differential Profile</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Regions</td>
<td>Cycles</td>
<td>HITM</td>
<td>RemDRAM</td>
</tr>
<tr>
<td>main</td>
<td>4.19X</td>
<td>13.2X</td>
<td>1.1X</td>
</tr>
<tr>
<td>2-4</td>
<td>5.34X</td>
<td>17.8X</td>
<td>1X</td>
</tr>
<tr>
<td>14-15</td>
<td>1.02X</td>
<td>1.03X</td>
<td>1X</td>
</tr>
<tr>
<td>15-16</td>
<td>1.03X</td>
<td>1.1X</td>
<td>1.01X</td>
</tr>
</tbody>
</table>
Prototypes for OpenMP and Task Parallelism

OMP-WHIP for OpenMP programs: [https://github.com/rutgers-apl/omp-whip/](https://github.com/rutgers-apl/omp-whip/)
TaskProf for Intel TBB programs: [https://github.com/rutgers-apl/TaskProf2/](https://github.com/rutgers-apl/TaskProf2/)

**Diagram:**
- **Input OpenMP program**
- **Compile**
- **OMPT Callback**
  - **OMP-WhIP library**
  - **Binary**
  - **Run**
  - **Parallelism profile**
  - **What-if regions**
  - **What-if profile**
  - **Differential profile**
Optimizing MILCmk

### Initial Parallelism Profile

<table>
<thead>
<tr>
<th>File:Line</th>
<th>Parallelism</th>
<th>Cpath</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>main</strong></td>
<td>44.21</td>
<td>28.3</td>
</tr>
<tr>
<td>vmeq.c:23</td>
<td>30.29</td>
<td>23.3</td>
</tr>
<tr>
<td>veq.c:28</td>
<td>32.83</td>
<td>19.55</td>
</tr>
<tr>
<td>vpeq.c:28</td>
<td>33.55</td>
<td>9.35</td>
</tr>
<tr>
<td>....</td>
<td>....</td>
<td>......</td>
</tr>
</tbody>
</table>

### What-if Profile

**What-if Regions**
- func.s.c:81 – 91
- func.s.c:60 – 67
- func.s.c:47 – 54

<table>
<thead>
<tr>
<th>File:Line</th>
<th>Parallelism</th>
<th>Cpath</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>main</strong></td>
<td>89.89</td>
<td>21.3</td>
</tr>
<tr>
<td>vmeq.c:23</td>
<td>30.29</td>
<td>25.2</td>
</tr>
<tr>
<td>veq.c:28</td>
<td>32.83</td>
<td>21.5</td>
</tr>
<tr>
<td>vpeq.c:28</td>
<td>33.55</td>
<td>11.5</td>
</tr>
<tr>
<td>....</td>
<td>....</td>
<td>......</td>
</tr>
</tbody>
</table>
Optimizing MILCmk

Replaced serial for loop with parallel_reduce

```c
QLA_Real
sum_V(QLA_ColorVector *d, int n)
{
    QLA_Real t=0, x=(QLA_Real *)d;
    int nn = n*sizeof(QLA_ColorVector)/sizeof(QLA_Real);
    for(int i=0; i<nn; i++) t += r[i];
    return t/nn;
}
```

```c
QLA_Real
sum_H(QLA_HalfFermion *d, int n)
{
    QLA_Real t=0, x=(QLA_Real *)d;
    int nn = n*sizeof(QLA_HalfFermion)/sizeof(QLA_Real);
    for(int i=0; i<nn; i++) t += r[i];
    return t/nn;
}
```

```c
QLA_Real
sum_D(QLA_DiracFermion *d, int n)
{
    QLA_Real t=0, x=(QLA_Real *)d;
    int nn = n*sizeof(QLA_DiracFermion)/sizeof(QLA_Real);
    for(int i=0; i<nn; i++) t += r[i];
    return t/nn;
}
```
## Optimizing MILCmk

### Initial Differential Profile

<table>
<thead>
<tr>
<th>File:Line</th>
<th>Differential Profile</th>
</tr>
</thead>
<tbody>
<tr>
<td>main</td>
<td>Cycles</td>
</tr>
<tr>
<td>veq.c:28-35</td>
<td>3.8X</td>
</tr>
<tr>
<td>vmeq.c:20-22</td>
<td>3.7X</td>
</tr>
<tr>
<td>vpeq.c:20-27</td>
<td>3.6X</td>
</tr>
</tbody>
</table>

- Inflation in cycles and remote DRAM accesses in 5 parallel_for regions
- parallel_for loops were repeated multiple times
  - Lack of affinity
- Optimized by replacing default partitioner with affinity partitioner

**Increased the speedup of MILCmk from 2.2X to 6X**
Is it Useful?

We found it to be effective with numerous applications.

Currently in talks for tech transfer with the Intel Vtune team.

Open Source at
https://github.com/rutgers-apl/TaskProf2
https://github.com/rutgers-apl/omp-whip/
Conclusion

• Make a case for measuring logical parallelism

• Series-parallel relations + fine-grained measurements ➞ a useful performance model for identifying scalability bottlenecks

• What-if analyses can help you identify regions that matter

• Differential analyses to identify regions having secondary effects

• Applicable to wide variety of programming models with appropriate series-parallel graphs
Develop Abstractions for Performance & Correctness

TaskProf2: [https://github.com/rutgers-apl/TaskProf2](https://github.com/rutgers-apl/TaskProf2)
OMP-WHIP: [https://github.com/rutgers-apl/omp-whip/](https://github.com/rutgers-apl/omp-whip/)
CASM-Verify: [https://github.com/rutgers-apl/CASM-Verify/](https://github.com/rutgers-apl/CASM-Verify/)

Other software prototypes from the Rutgers Architecture & Programming Languages Group:
[https://github.com/rutgers-apl/](https://github.com/rutgers-apl/)