Kremlin: Rethinking and Rebooting \texttt{gprof} for the Multicore Era

Saturnino Garcia, Donghwan Jeon, Chris Louie, Michael B. Taylor

Computer Science & Engineering Department
University of California, San Diego
Motivating a “gprof for parallelization”

How effective are programmers at picking the right parts of a program to parallelize?

User study* we performed at UC San Diego (UCSD IRB #100056)

First and second year CS graduate students

Users parallelize their programs and submit to job queue for timing

32-core AMD machine, Cilk++, access to gprof

Students were graded based on effectiveness of their parallel speedup; students told serial optimization would not help their grade

*Disclaimer: No graduate students were
User Study: Results

Examined student’s activities to determine result of efforts

• Significant fraction of fruitless effort because of three basic problems

1. Low Parallelism: Region was not parallel enough
2. Low Coverage: Region’s execution time was too small
3. Poor Planning: Speedup negated by subsequent parallelization

User 143
User 249
User 371

Fruitless Parallelization Effort
**gprof** answers the question:

“What parts of this program should I spend time **optimizing**?”

**Kremlin** answers the question:

“What parts of this program should I spend time **parallelizing**?”
**Kremlin’s Usage Model**

Usage model inspired by gprof

1. Compile instrumented binary
2. Profile with sample input
3. Run analysis tool to create plan

```bash
$> make CC=kremlin-cc
$> ./tracking lolcats
$> kremlin tracking --personality=openmp
```

<table>
<thead>
<tr>
<th>File (lines)</th>
<th>Self-P</th>
<th>Cov (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>imageBlur.c (49-58)</td>
<td>145.3</td>
<td>9.7</td>
</tr>
<tr>
<td>imageBlur.c (37-45)</td>
<td>145.3</td>
<td>8.7</td>
</tr>
<tr>
<td>getInterpPatch.c (26-35)</td>
<td>25.3</td>
<td>8.9</td>
</tr>
<tr>
<td>calcSobel_dX.c (59-68)</td>
<td>126.2</td>
<td>8.1</td>
</tr>
<tr>
<td>calcSobel_dX.c (46-55)</td>
<td>126.2</td>
<td>8.1</td>
</tr>
</tbody>
</table>
Kremlin’s Key Components

- **Hierarchical Critical Path Analysis (HCPA)**
  - Quantifies self-parallelism in each program region
- **Self-Parallelism**
  - Estimates ideal parallel speedup of a specific region

---

**Parallelism Discovery**

“What’s the potential parallel speedup of each part of this program?”

---

**Parallelism Planning**

“What regions must I parallelize to get the maximum benefit on this system?”
Developing an Approach for Parallelism Discovery

Existing Technique: 1980’s-era Critical Path Analysis (CPA)

Finds critical path through the dynamic execution of a program

Mainly used in research studies to quantify limits of parallelism

parallelism = \frac{\text{work}}{\text{critical path length}}

work \sim \# \text{ of instrs}

- instruction
- data or control dependence

\text{critical path (cp)}
Benefits of CPA as a Basis for a Parallelism Discovery

Evaluates program’s potential for parallelization under relatively optimistic assumptions

† Closer approximation to what human experts can achieve versus pessimistic static analysis in automatic parallelizing compilers

† It is predictive of parallelism after typical parallelization transformations

† E.g., Loop interchange, loop fission, locality enhancement
Improving CPA with Hierarchical CPA (HCPA)

A is typically run on an entire program
Not helpful for identifying specific regions to parallelize
Doesn’t help evaluate execution time of a program if only a subset of the program is parallelized

Hierarchical CPA is a region-based analysis
Self-Parallelism (sp) identifies parallelism in specific regions
Provides basis for estimating parallel speedup of individual regions

```plaintext
for (i=1..100) {
    for (j=1..100) {
        a[i][j] = a[i][j-1]+3;
        b[i][j] = b[i][j-1]+5;
    }
}
```

sp=2  sp=1  sp=100
HCPA Step 1: Hierarchically Apply CPA

**Goal:** Introduce localization through *region*-based analysis

Shadow-memory based implementation

- Performs CPA analysis on every program region
- Single pass: Concurrently analyzes multiple nested regions

```
1..100) {
    for (j=1..100) {
        a[i][j] = a[i][j-1]+3;
        b[i][j] = b[i][j-1]+5;
    }
}
```

\[ p = \frac{W}{CP} = \frac{100000}{500} = 200 \times \]
HCPA Step 2: Calculate Self-Parallelism

Goal: Eliminate effect of nested parallelism in parallelism calculation

Approximate self-parallelism using only HCPA output

“Subtracts” nested parallelism from overall parallelism

\[
\text{work(for}_i) = 100 \times \text{work(for}_j) \\
\text{work'}(for_i) = 100 \times \text{CP(for}_j)
\]

\[
p = \frac{W}{CP} = \frac{100000}{500} = 200
\]

\[
W' = 100 \times 500
\]
**PA Step 3: Compute Static Region Data**

**Goal:** Convert dynamic region data to static region output

- Merge dynamic nodes associated with same static region

**Work:** *Sum* of work across dynamic instances

**Self-Parallelism:** *Weighted Average* across dynamic instances

\[
\text{for}(i) \quad (\text{work, sp}) = (100000, 100) \\
\text{...} \\
\text{for}(i) \quad (100000, 1) \\
\text{...} \\
\text{for}(i) \quad (100000, 2)
\]

Merge dynamic regions
Further Details on Discovery in Our Paper

Kremlin handles much more complex structures than just nested for loops: finds parallelism in arbitrary code including recursion.

Self-parallelism metric is defined and discussed in detail in the paper.

Compression technique used to reduce size of HCPA output.
Creating a Parallelization Plan

**Goal:** Use HCPA output to select best regions for target system

**Planning personalities** allow user to incorporate system constraints

- **Software constraints:** What types of parallelism can I specify?
- **Hardware constraints:** Synchronization overhead, etc.

Planning algorithm can change based on constraints
An OpenMP Planner

Based on OpenMP 2.0 specification

Focused on loop-level parallelism

Disallows nested parallelism because of overhead

Planning algorithm based on dynamic programming

parallelized time reduction = W - (W/SP)

<table>
<thead>
<tr>
<th>Region</th>
<th>Work</th>
<th>SP</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>100k</td>
<td>2</td>
</tr>
<tr>
<td>B</td>
<td>50k</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>50k</td>
<td>10</td>
</tr>
<tr>
<td>D</td>
<td>50k</td>
<td>10</td>
</tr>
<tr>
<td>E</td>
<td>25k</td>
<td>5</td>
</tr>
</tbody>
</table>
Methodology:

Ran Kremlin on serial versions; targeting OpenMP
Parallelized according to Kremlin’s plan
Gathered performance results on 8 socket AMD 8380 Quad-core
Compared against third-party parallelized versions (3rd Party)

Benchmarks: NAS OpenMP and SpecOMP
Have both serial and parallel versions
Wide range of parallel speedup (min: 1.85x, max: 25.89x) on 32 cores
How much effort is saved using Kremlin?

<table>
<thead>
<tr>
<th>Suite</th>
<th>Benchmark</th>
<th>3rd Party</th>
<th>Kremlin</th>
<th>Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>ecOMP</td>
<td>art</td>
<td>3</td>
<td>4</td>
<td>0.75x</td>
</tr>
<tr>
<td></td>
<td>ampp</td>
<td>6</td>
<td>3</td>
<td>2.00x</td>
</tr>
<tr>
<td></td>
<td>equake</td>
<td>10</td>
<td>6</td>
<td>1.67x</td>
</tr>
<tr>
<td>NPB</td>
<td>ep</td>
<td>1</td>
<td>1</td>
<td>1.00x</td>
</tr>
<tr>
<td></td>
<td>is</td>
<td>1</td>
<td>1</td>
<td>1.00x</td>
</tr>
<tr>
<td></td>
<td>ft</td>
<td>6</td>
<td>6</td>
<td>1.00x</td>
</tr>
<tr>
<td></td>
<td>mg</td>
<td>10</td>
<td>8</td>
<td>1.25x</td>
</tr>
<tr>
<td></td>
<td>cg</td>
<td>22</td>
<td>9</td>
<td>2.44x</td>
</tr>
<tr>
<td></td>
<td>lu</td>
<td>28</td>
<td>11</td>
<td>2.55x</td>
</tr>
<tr>
<td></td>
<td>bt</td>
<td>54</td>
<td>27</td>
<td>2.00x</td>
</tr>
<tr>
<td></td>
<td>sp</td>
<td>70</td>
<td>58</td>
<td>1.21x</td>
</tr>
<tr>
<td>Overall</td>
<td></td>
<td>211</td>
<td>134</td>
<td>1.57x</td>
</tr>
</tbody>
</table>
How good is Kremlin-guided performance?

Compared performance against expert, third-party version.

**Significantly better results on two benchmarks**

Required 65 fewer regions to get within 4% of performance on others (1.87X improvement)
Does Kremlin pick the best regions first?

Determined what % of speedup comes from first \{25,50,75,100\}% of recommended regions

<table>
<thead>
<tr>
<th>Fraction of Kremlin Plan Applied</th>
<th>First 25% of regions</th>
<th>Second 25% of regions</th>
<th>Third 25% of regions</th>
<th>Last 25% regions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Marginal benefit (% max speedup) (avg)</td>
<td>56.2%</td>
<td>30.2%</td>
<td>9.2%</td>
<td>4.4%</td>
</tr>
</tbody>
</table>

86.4% in first half, decreasing marginal benefit
Conclusion

Kremlin helps a programmer determine: *What parts of this program should I spend time parallelizing?*

Three key techniques introduced by Kremlin

*Hierarchical CPA*: How much total parallelism is in each region?

*Self-Parallelism*: How much parallelism is only in this region?

*Planning Personalities*: What regions are best for my target system?

Compelling results

- 1.57x average reduction in number of regions parallelized
- Greatly improved performance on 2 of 11 benchmarks; very close on others
## Parallelism for Three Common Loop Types

<table>
<thead>
<tr>
<th>Loop Type</th>
<th>DOALL</th>
<th>DOACROSS</th>
<th>Serial</th>
</tr>
</thead>
<tbody>
<tr>
<td>Work (ET')</td>
<td>$N \times CP$</td>
<td>$(N/2) \times CP$</td>
<td>$N \times CP$</td>
</tr>
<tr>
<td>CP</td>
<td>$CP$</td>
<td>$CP$</td>
<td>$CP$</td>
</tr>
<tr>
<td>Critical Path (CP)</td>
<td>$CP$</td>
<td>$CP$</td>
<td>$CP$</td>
</tr>
<tr>
<td>Length (CP)</td>
<td>$N$</td>
<td>$2.0$</td>
<td>$1.0$</td>
</tr>
</tbody>
</table>

### Equations

- $\frac{N \times CP}{CP} = N$
- $\frac{N \times CP}{(N/2) \times CP} = 2.0$
- $\frac{N \times CP}{N \times CP} = 1.0$
Kremlin System Architecture

Source Code

Static Instrumentation

Critical Path Instrumentation

Region Instrumentation

Instrumented Binary

KremLib

Parallelism Profile

A: <p,w>
B: <p,w>
C: <p,w>
...

Kremlin Planner

Region Graph

Ordered Parallelism Plan

exec w/ inputs
Interpreting the Parallelism Metric

- **Totally Serial**
  - All work is on critical path \((ET == CP)\)

- **Highly Parallel**
  - Most work is off critical path \((ET >> CP)\)

Parallelism is a result of execution time.