PCERE: Fine-grained Parallel Benchmark Decomposition for Scalability Prediction

Mihail Popov, Chadi Akel, Florent Conti, William Jalby, Pablo de Oliveira Castro

UVSQ - PRiSM - ECR

Mai 28, 2015
Evaluate strong scalability

- Evaluate strong scalability of OpenMP applications is costly and time-consuming
- Execute multiple times the whole application with different thread configurations
- Waste of resources
  - According to Amdahl’s law sequential parts do not scale
  - Parallel regions may share similar performance across invocations
**PCERE: Parallel Codelet Extractor and REplayer**

- **Accelerate strong scalability evaluation** with PCERE
- PCERE is part of CERE (Codelet Extractor and REplayer) framework
- Decompose applications into small pieces called *Codelets*
- Each codelet maps a parallel region and is a standalone executable
- Extract codelets once
- Replay codelets instead of applications with different number of threads
int main()
{
    for(i=0; i<3; i++){
        // sequentiel code
        #pragma omp parallel A
    }
    // sequentiel code
    #pragma omp parallel B
    // sequentiel code
}

Executing the whole application with different threads configurations
Directly replaying the parallel regions

Extracting parallel regions A and B and measuring sequentiel execution time

int main()
{
  for(i=0;i<3;i++){
    //sequentiel code
    #pragma omp parallel A
  }
  //sequentiel code
  #pragma omp parallel B
  //sequentiel code
}
int main()
{
for(i=0;i<3;i++){
  //sequentiel code
  #pragma omp parallel A
} 
//sequentiel code ...
//sequentiel code ...
}

Add sequential time and parallel region multiple invocations
Outline

1 Overview

2 Extract and replay codelets

3 Prediction model evaluation
Overview

Codelet capture and replay

- OpenMP Applications
- Parallel region outlining
- Capture of representative working sets
- Region Capture
  - Change number of threads or affinity
  - Working sets memory dump
- Codelet Replay
  - Fast performance prediction
  - Warmup + Replay
  - Generate codelets wrapper
  - Retarget for different architecture
LLVM OpenMP Intermediate Representation extraction

- Extract codelets at Intermediate Representation for language portability and cross architecture evaluation
Clang front end transforms source code into IR

C code

```c
void main()
{
    #pragma omp parallel
    {
        int p = omp_get_thread_num();
        printf("%d", p);
    }
}
```

Clang OpenMP front end

```c
define i32 @main() {
    entry:
    ...
    call @__kmpc_fork_call @.omp_microtask.(...)
    ...
}
```

define internal void @.omp_microtask.(...) {
    entry:
    %p = alloca i32, align 4
    %call = call i32 @omp_get_thread_num()
    store i32 %call, i32* %p, align 4
    %1 = load i32* %p, align 4
    call @printf(%1)
}

LLVM simplified IR

Thread execution model
Deterministic codelet replay

Parallel region capture

Parallel region replay

Dump call

Direct jump to parallel region

Restore call

Exit
Memory dump

- System memory snapshot at the beginning of each parallel region

---

define i32 @main() {
  entry:
  ...
call @__kmvc_fork_call @.omp_microtask.(...)
  ...
}
define internal void @.omp_microtask.(...) {
  entry:
  %p = alloca i32, align 4
  %call = call i32 @omp_get_thread_num()
  store i32 %call, i32* %p, align 4
  %1 = load i32* %p, align 4
  call @printf(%1)
}

---

LLVM simplified IR

define i32 @main() {
  entry:
  ...
call @__extracted__.omp_microtask.(...)
  ...
}
define internal void @__extracted__.omp_microtask.(...) {
  newFuncRoot:
  call void @dump(...)
call @__kmvc_fork_call @ .omp_microtask.(...)
}
define internal void @.omp_microtask.(...) {
  entry:
  ...
}

LLVM simplified IR

extract + dump passes
Extract and replay codelets

**Codelet replay**

- Reload codelet working set
- Reproduce cache state with optimistic cache warm-up
- Multiple working sets for a single codelet
Codelets with different working sets

Figure: MG resid execution time over the different invocations replayed with 4 threads
Lock Support

- Lock support on Linux uses Futex
- Each futex allocates a kernel space wait queue
- Memory capture saves only the user space memory
- Lock capture step that detects all the locks accessed by a codelet
- Replay wrapper initialize required locks in kernel space
Test benchmarks and architectures

- Using NAS Parallel Benchmark OpenMP 3.0 C version based on the Omni Compiler Project

<table>
<thead>
<tr>
<th></th>
<th>Core2</th>
<th>Nehalem</th>
<th>Sandy Bridge</th>
<th>Ivy Bridge</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>E7500</td>
<td>Xeon E5620</td>
<td>E5</td>
<td>i7-3770</td>
</tr>
<tr>
<td>Frequency (GHz)</td>
<td>2.93</td>
<td>2.40</td>
<td>2.7</td>
<td>3.4</td>
</tr>
<tr>
<td>Sockets</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Cores per socket</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>Threads per core</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>L1 cache (KB)</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>L2 cache (KB)</td>
<td>3MB</td>
<td>256</td>
<td>256</td>
<td>256</td>
</tr>
<tr>
<td>L3 cache (MB)</td>
<td>-</td>
<td>12</td>
<td>20</td>
<td>8</td>
</tr>
<tr>
<td>Ram (GB)</td>
<td>4</td>
<td>24</td>
<td>64</td>
<td>32</td>
</tr>
</tbody>
</table>

Figure: Test architectures
Reproducing parallel regions scaling with codelets

**Figure**: Real vs. PCERE execution time predictions on Sandy Bridge for the SP compute rhs codelet.
## Prediction accuracy

<table>
<thead>
<tr>
<th></th>
<th>BT</th>
<th>EP</th>
<th>LU</th>
<th>FT</th>
<th>SP</th>
<th>CG</th>
<th>IS</th>
<th>MG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core2</td>
<td>2.4</td>
<td>0.1</td>
<td>1.4</td>
<td>2.3</td>
<td>1.8</td>
<td>1</td>
<td>4.2</td>
<td>1.5</td>
</tr>
<tr>
<td>Nehalem</td>
<td>3</td>
<td>0.4</td>
<td>0.6</td>
<td>0.5</td>
<td>6</td>
<td>9.8</td>
<td>2.2</td>
<td>0.6</td>
</tr>
<tr>
<td>Sandy Bridge</td>
<td>8.9</td>
<td>1.5</td>
<td>1.4</td>
<td>1.6</td>
<td>0.9</td>
<td>14.5</td>
<td>18</td>
<td>12.1</td>
</tr>
<tr>
<td>Ivy Bridge</td>
<td>0.7</td>
<td>1</td>
<td>2</td>
<td>3.4</td>
<td>1.2</td>
<td>3.7</td>
<td>5.3</td>
<td>5.4</td>
</tr>
</tbody>
</table>

**Figure**: NAS 3.0 C version average percentage error prediction accuracy

- **On Ivy Bridge**, PCERE predicts FT execution time scalability with an error of 3.4%
## Benchmarking acceleration

<table>
<thead>
<tr>
<th></th>
<th>BT</th>
<th>EP</th>
<th>LU</th>
<th>FT</th>
<th>SP</th>
<th>CG</th>
<th>IS</th>
<th>MG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core2</td>
<td>31.5</td>
<td>1</td>
<td>54.3</td>
<td>1.5</td>
<td>87.2</td>
<td>24.2</td>
<td>1.1</td>
<td>0.9</td>
</tr>
<tr>
<td>Nehalem</td>
<td>43.2</td>
<td>1</td>
<td>51.9</td>
<td>2.1</td>
<td>97</td>
<td>21.1</td>
<td>1.2</td>
<td>2.2</td>
</tr>
<tr>
<td>Sandy Bridge</td>
<td>45.5</td>
<td>1</td>
<td>44.6</td>
<td>2.4</td>
<td>79</td>
<td>13.2</td>
<td>1.2</td>
<td>2.4</td>
</tr>
<tr>
<td>Ivy Bridge</td>
<td>39</td>
<td>1</td>
<td>45.5</td>
<td>2.1</td>
<td>82</td>
<td>17</td>
<td>1.1</td>
<td>1.8</td>
</tr>
</tbody>
</table>

**Figure**: NAS 3.0 C version average benchmarking acceleration

- **On Core2**, PCERE CG scalability evaluation is 24.2 times faster than with normal executions.
## PCERE prediction accuracy and benchmarking acceleration

<table>
<thead>
<tr>
<th></th>
<th>Core2</th>
<th>Nehalem</th>
<th>Sandy Bridge</th>
<th>Ivy Bridge</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Accuracy</strong></td>
<td>1.8%</td>
<td>2.9%</td>
<td>7.4%</td>
<td>2.8%</td>
</tr>
<tr>
<td><strong>Acceleration</strong></td>
<td>25.2</td>
<td>27.4</td>
<td>23.7</td>
<td>23.7</td>
</tr>
</tbody>
</table>

**Figure**: NAS 3.0 C version average **prediction accuracy** and **benchmarking acceleration** per architecture
Cross micro-architecture codelet replay

- Capture-Replay is micro-architecture agnostic
- Capture on Nehalem → Replay on Sandy Bridge

<table>
<thead>
<tr>
<th>Threads</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accuracy</td>
<td>3</td>
<td>2.3</td>
<td>3</td>
<td>7.8</td>
<td>11.5</td>
<td>17.6</td>
</tr>
</tbody>
</table>

Figure: NAS 3.0 C version **average percentage error** cross replay accuracy

<table>
<thead>
<tr>
<th>Application</th>
<th>BT</th>
<th>EP</th>
<th>LU</th>
<th>FT</th>
<th>SP</th>
<th>CG</th>
<th>IS</th>
<th>MG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accuracy</td>
<td>9.8</td>
<td>0.3</td>
<td>1.1</td>
<td>3.8</td>
<td>3.9</td>
<td>18.1</td>
<td>6.4</td>
<td>17</td>
</tr>
</tbody>
</table>

Figure: NAS 3.0 C version **average percentage error** cross replay accuracy
Limitation and future work

- **Limitations**
  - No acceleration on applications with a single parallel region and no relevant sequential parts (EP)
  - Prediction error due to variant sequential time across thread configurations (IS)

- **Future work**
  - Improve warm-up strategy: use CERE page traces warm-up
  - Apply a clustering approach over codelets
  - OpenMP parameters space exploration with codelets
Conclusion

- To be released with CERE at http://benchmark-subsetting.github.io/pcere/
- Extract codelets once, replay them many times
- Cross micro-architecture and thread configuration extraction and replay
- Accelerate strong scalability evaluation 25 times
- Strong scalability prediction average error of 3.7%
Codelet replay

- Optimistic cache warm-up: assuming that the codelet working set is hot in the original run

```
void main()
{
    int i;
    int iteration = 1;
    for(i=0;i<iteration;i++)
        run__extracted__.omp_microtask();
}
```

***Updated main C code***

```
define void @run_extracted__.omp_microtask() {
    entry:
    call void @load(...) 
    ... %Arrange arguments 
    call @__extracted__.omp_microtask(...) 
}
```

***LLVM simplified IR***

```
define internal void @__extracted__.omp_microtask(...) {
    newFuncRoot:
    call @__kmpc_fork_call @.omp_microtask(...) 
}
```

M.Popov C.Akel F.Conti W.Jalby P.Oliveira

PCERE

Mai 28, 2015 18 / 18
Related work

- Cross-platform performance prediction of parallel applications using partial execution. Yang, Leo T and Ma, Xiaosong and Mueller, Frank SC 2005
- Detecting Phases in Parallel Applications on Shared Memory Architectures. Perelman, Erez and Polito, Marzia and Bouguet, J-Y and Sampson, Jack and Calder, Brad and Dulong, Carole IPDPS 2006
- BarrierPoint: Sampled Simulation of Multi-Threaded Applications. Carlson, Trevor E and Heirman, Wim and Van Craeynest, Kenzo and Eeckhout, Lieven ISPASS 2014
- Effective source-to-source outlining to support whole program empirical optimization. Liao, Chunhua and Quinlan, Daniel J and Vuduc, Richard and Panas, Thomas Languages and Compilers for Parallel Computing 2010
void main()
{
    (...)
    Loop A
    Loop B
    (...)
}

CERE
Loops IR extraction
with no optimization
Intermediate representation
Loops IR extraction
with no optimization
Compile with an optimization point
Optimization space to explore

Loops profiling and extraction
With -O2
Representative invocations working sets
Prediction model
Fast optimization point evaluation
Codelet optimization and replay

Loops profiling and extraction
With -O2
Representative invocations working sets
Prediction model
Fast optimization point evaluation
Codelet optimization and replay
Flags exploration

- For each optimization sequence, only replay the relevant parts
- Codelets matching over 200 optimization sequences

<table>
<thead>
<tr>
<th>Application</th>
<th>Median error</th>
<th>Average error</th>
</tr>
</thead>
<tbody>
<tr>
<td>CG</td>
<td>2.0</td>
<td>3.0</td>
</tr>
<tr>
<td>EP</td>
<td>0.1</td>
<td>0.3</td>
</tr>
<tr>
<td>FT</td>
<td>3.6</td>
<td>5.0</td>
</tr>
<tr>
<td>IS</td>
<td>1.3</td>
<td>4.8</td>
</tr>
<tr>
<td>LU</td>
<td>1.2</td>
<td>2.2</td>
</tr>
<tr>
<td>MG</td>
<td>1.9</td>
<td>3.4</td>
</tr>
<tr>
<td>SP</td>
<td>2.1</td>
<td>2.6</td>
</tr>
<tr>
<td>RTM</td>
<td>5.3</td>
<td>5.8</td>
</tr>
</tbody>
</table>

Figure: Matching error percentage per application

- Speed-up evaluation versus matching error
  - -O2 RTM evaluation is 237 times cheaper with codelets