The Cilkprof Scalability Profiler

Tao B. Schardl  Bradley C. Kuszmaul  I-Ting Angelina Lee*
William M. Leiserson  Charles E. Leiserson

MIT Computer Science and Artificial Intelligence Laboratory

*Washington University in St. Louis

SPAA 2015
Quicksort

C++ quicksort:

```cpp
void qsort(int64_t array[], size_t n,
           size_t l, size_t h) {
  // ... base case ...
  size_t part;
  part = partition(array, n, l, h);
  qsort(array, n, l, part);
  qsort(array, n, part, h);
}

int main(int argc, char* argv[]) {
  // ... initialization ...
  qsort(array, n, 0, n);
  // ... use array ...
  return 0;
}
```
Parallel quicksort using Cilk

The `cilk_spawn` and `cilk_sync` keywords expose parallel work.

C++ quicksort:

```cpp
def qsort(int64_t array[], size_t n,
    size_t l, size_t h) {
  // ... base case ...
  size_t part;
  part = partition(array, n, l, h);
  qsort(array, n, l, part);
  qsort(array, n, part, h);
}
```

```cpp
int main(int argc, char* argv[]) {
  // ... initialization ...
  qsort(array, n, 0, n);
  // ... use array ...
  return 0;
}
```

Cilk parallel quicksort:

```cpp
def qsort(int64_t array[], size_t n,
    size_t l, size_t h) {
  // ... base case ...
  size_t part;
  part = partition(array, n, l, h);
  cilk_spawn qsort(array, n, l, part);
  qsort(array, n, part, h);
  cilk_sync;
}
```

```cpp
int main(int argc, char* argv[]) {
  // ... initialization ...
  qsort(array, n, 0, n);
  // ... use array ...
  return 0;
}
```
Parallel quicksort using Cilk

The **cilk_spawn** and **cilk_sync** keywords expose parallel work.

- The **cilk_spawn** allows the two recursive **qsort** calls to execute in parallel.
- The **cilk_sync** waits for both recursive **qsort** calls return.

Cilk parallel quicksort:

```c
void qsort(int64_t array[], size_t n,
           size_t l, size_t h) {
    // ... base case ...
    size_t part;
    part = partition(array, n, l, h);
    cilk_spawn qsort(array, n, l, part);
    qsort(array, n, part, h);
    cilk_sync;
}

int main(int argc, char* argv[]) {
    // ... initialization ...
    qsort(array, n, 0, n);
    // ... use array ...
    return 0;
}
```

Schardl et al. The Cilkprof Scalability Profiler SPAA 2015 3 / 34
Running Cilk parallel quicksort

The parallel quicksort code can be compiled and run similarly to its serial counterpart.

Using ICC:

```bash
$ icpc -O3 qsort.cpp -o qsort
$ ./qsort -n 100000000
```

Using GCC:

```bash
$ g++ -O3 qsort.cpp -o qsort -fcilkplus
$ ./qsort -n 100000000
```

Using Cilk Plus/LLVM:

```bash
$ clang++ -O3 qsort.cpp -o qsort -fcilkplus -ldl
$ ./qsort -n 100000000
```
Running Cilk parallel quicksort

The parallel quicksort code can be compiled and run similarly to its serial counterpart.

Using ICC:

```
$ icpc -O3 qsort.cpp -o qsort
$ ./qsort -n 100000000
```

Using GCC:

```
$ g++ -O3 qsort.cpp -o qsort -fcilkplus
$ ./qsort -n 100000000
```

Using Cilk Plus/LLVM:

```
$ clang++ -O3 qsort.cpp -o qsort -fcilkplus -ldl
$ ./qsort -n 100000000
```

Questions: How fast is this code? Does it speed up on multiple processors? How parallel is it?
Cilkview [HLL10] analyzes the scalability of a Cilk program.

$ icpc -O3 qsort.cpp -o qsort$
$ cilkview --trials=all -- ./qsort -n 100000000$

- Cilkview instruments the executable binary of the Cilk program (with a little help from ICC).
- Cilkview executes the program serially while keeping track of the logical series-parallel relationships between instructions.
- Cilkview incurs only a constant-factor slowdown over the program’s serial running time.
Cilkview’s output for qsort(100M)

Note: This output has been simplified for didactic purposes.

Work: 36,272,478,614 instructions
Span: 1,621,934,437 instructions
Parallelism: 22.36
Cilkview models the Cilk program performance as follows:

- Let $T_P$ denote the execution time on $P$ processors.
- **Work** is serial execution time $T_1$.
- **Speedup** on $P$ processors is $T_1/T_P \leq P$.
  - **Linear speedup** occurs when $T_1/T_P = P$.
- **Span** is critical-path length $T_\infty$.
- **Parallelism** is $T_1/T_\infty$, the maximum possible speedup.

As a practical matter, a Cilk program should exhibit **parallel slackness** of 10 — it should exhibit $\geq 10P$ parallelism.
Cilkview tells us that \texttt{qsort} does not exhibit much parallelism.

Work, span, and parallelism of \texttt{qsort}(100M)

- Work: 36,272,478,614 instructions
- Span: 1,621,934,437 instructions
- Parallelism: 22.36
where is the bottleneck?

Cilkview does not tell us where the scalability bottleneck is in `qsort`.

```c
void qsort(int64_t array[], size_t n,
           size_t l, size_t h) {
    // ... base case ...
    size_t part;
    part = partition(array, n, l, h);
    cilk_spawn qsort(array, n, l, part);
    qsort(array, n, part, h);
    cilk_sync;
}

int main(int argc, char* argv[]) {
    // ... initialization ...
    qsort(array, n, 0, n);
    return 0;
}
```

Speedup of `qsort`
Our contribution: Cilkprof

Cilkprof profiles the parallelism of a Cilk program.

$ clang++ -O3 -g qsort.cpp -o qsort -fcilkplus -ldl -fcilktool-instr-c -lcilkprof
$ ./qsort -n 100000000

- We modified the Cilk Plus/LLVM compiler to instrument functions, spawns, and syncs in a Cilk program.
- We implemented Cilkprof as a library to link into an instrumented Cilk program.
- Running the instrumented program linked with Cilkprof produces a spreadsheet attributing portions of the program’s work and span to different call sites.
  - *No user interface*
Cilkprof’s output for \texttt{qsort(100M)}

Cilkprof gathers work and span measurements for every call site.

<table>
<thead>
<tr>
<th>File</th>
<th>Line</th>
<th>Top-caller work on work</th>
<th>Local work on work</th>
<th>Top-caller span on span</th>
<th>Local span on span</th>
</tr>
</thead>
<tbody>
<tr>
<td>qsort.cpp</td>
<td>5</td>
<td>0.6</td>
<td>10.4</td>
<td>0.6</td>
<td>3.3</td>
</tr>
<tr>
<td>qsort.cpp</td>
<td>6</td>
<td>1.5</td>
<td>3.2</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>qsort.cpp</td>
<td>7</td>
<td>14.3</td>
<td>2.7</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>qsort.cpp</td>
<td>13</td>
<td>16.3</td>
<td>0.0</td>
<td>3.3</td>
<td>0.0</td>
</tr>
</tbody>
</table>
Cilkprof’s output for qsort(100M)

Cilkprof gathers work and span measurements for every call site.

<table>
<thead>
<tr>
<th></th>
<th>On work (gigacycles)</th>
<th></th>
<th>On span (gigacycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Top-caller work</td>
<td>Local work</td>
<td>Top-caller span</td>
</tr>
<tr>
<td>void qsort(/<em>...</em>/)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>/* ... base case ... */</td>
<td>0.6 10.4</td>
<td>0.6 3.3</td>
<td></td>
</tr>
<tr>
<td>part = partition(/<em>...</em>/);</td>
<td>1.5 3.2</td>
<td>0.0 0.0</td>
<td></td>
</tr>
<tr>
<td>cilk_spawn qsort(/<em>...</em>/);</td>
<td>14.3 2.7</td>
<td>2.8 0.0</td>
<td></td>
</tr>
<tr>
<td>qsort(/<em>...</em>/);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cilk_sync;</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

01-06

int main(/*...*/) {
  // ... initialization ...
  qsort(/*...*/); 16.3 0.0 3.3 0.0
  // ... use array ...
09-13

Schardl et al.
The Cilkprof Scalability Profiler
SPAA 2015 11 / 34
Compiler instrumentation

We used compiler instrumentation for its efficiency.

<table>
<thead>
<tr>
<th>Sampling</th>
<th>Binary</th>
<th>Compiler</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Example tools</strong></td>
<td>gprof, pprof, perf</td>
<td>valgrind, DynamoRio,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Cilkview (Via Pin)</td>
</tr>
<tr>
<td><strong>Overhead</strong></td>
<td>✓  None</td>
<td>×  Lots</td>
</tr>
<tr>
<td><strong>Properties</strong></td>
<td>✓  No recompilation</td>
<td>✓  No recompilation</td>
</tr>
<tr>
<td></td>
<td>necessary.</td>
<td>necessary.</td>
</tr>
<tr>
<td></td>
<td>×  Statistical; don’t</td>
<td>~  Creates measurement</td>
</tr>
<tr>
<td></td>
<td>know how to measure span</td>
<td>error, but can still</td>
</tr>
<tr>
<td></td>
<td>via sampling.</td>
<td>measure parallelism</td>
</tr>
<tr>
<td></td>
<td></td>
<td>well enough.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>~  Requires recompilation.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>~  Creates measurement</td>
</tr>
<tr>
<td></td>
<td></td>
<td>error, but can still</td>
</tr>
<tr>
<td></td>
<td></td>
<td>measure parallelism</td>
</tr>
<tr>
<td></td>
<td></td>
<td>well enough.</td>
</tr>
</tbody>
</table>

Schardl et al. The Cilkprof Scalability Profiler
Cilkprof implements an efficient serial algorithm.

- Cilkprof profiles a Cilk program with work $T_1$ in $\Theta(T_1)$ time.
  - *Constant overhead*, independent of number of call sites.

- Remarkably, in the same asymptotic time Cilkview takes to measure work and span for the whole program, Cilkprof measures work and span for *every call site*.

- For video encoding, in the asymptotic time Cilkview takes to measure two values, Cilkprof can measure those values for $\approx 3000$ call sites.
Cilkprof’s empirical performance

Cilkprof is efficient in practice.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>mm</td>
<td>Matrix multiplication</td>
<td>0.99</td>
</tr>
<tr>
<td>dedup</td>
<td>Compression</td>
<td>1.03</td>
</tr>
<tr>
<td>lu</td>
<td>LU decomposition</td>
<td>1.04</td>
</tr>
<tr>
<td>strassen</td>
<td>Strassen</td>
<td>1.06</td>
</tr>
<tr>
<td>heat</td>
<td>Heat diffusion</td>
<td>1.07</td>
</tr>
<tr>
<td>cilksort</td>
<td>Mergesort</td>
<td>1.08</td>
</tr>
<tr>
<td>pbfs</td>
<td>Breadth-first search</td>
<td>1.10</td>
</tr>
<tr>
<td>fft</td>
<td>Fast Fourier transform</td>
<td>1.15</td>
</tr>
<tr>
<td>quicksort</td>
<td>Quicksort</td>
<td>1.20</td>
</tr>
<tr>
<td>nqueens</td>
<td>(n)-Queens</td>
<td>1.27</td>
</tr>
<tr>
<td>ferret</td>
<td>Image similarity</td>
<td>2.04</td>
</tr>
<tr>
<td>leiserchess</td>
<td>Game-tree search</td>
<td>3.72</td>
</tr>
<tr>
<td>collision</td>
<td>Collision detection</td>
<td>4.37</td>
</tr>
<tr>
<td>cholesky</td>
<td>Cholesky decomposition</td>
<td>4.54</td>
</tr>
<tr>
<td>hevc</td>
<td>H265 video coding</td>
<td>6.25</td>
</tr>
<tr>
<td>fib</td>
<td>Fibonacci</td>
<td>7.36</td>
</tr>
</tbody>
</table>

Cilkprof incurs the following overheads:

- A geometric-mean slowdown of \(1.9\times\).
- A maximum slowdown of \(7.4\times\).

Cilkprof performs favorably to similar debugging tools.
Outline

1. Case study: qsort
2. Case study: pbfs
3. Profiling the work and span
4. Conclusion
Outline

1. Case study: qsort
2. Case study: pbfs
3. Profiling the work and span
4. Conclusion
Cilkprof’s profile for qsort

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
<th>On work (gigacycles)</th>
<th>On span (gigacycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Top-caller</td>
<td>Local work</td>
</tr>
<tr>
<td>01</td>
<td>void qsort(/<em>...</em>/) {</td>
<td></td>
<td></td>
</tr>
<tr>
<td>02</td>
<td>/* ... base case ... */</td>
<td></td>
<td></td>
</tr>
<tr>
<td>03</td>
<td>part = partition(/<em>...</em>/);</td>
<td>0.6</td>
<td>10.4</td>
</tr>
<tr>
<td>04</td>
<td>cilk_spawn qsort(/<em>...</em>/);</td>
<td>1.5</td>
<td>3.2</td>
</tr>
<tr>
<td>05</td>
<td>qsort(/<em>...</em>/);</td>
<td>14.3</td>
<td>2.7</td>
</tr>
<tr>
<td>06</td>
<td>cilk_sync;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>07</td>
<td>}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>08</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>09</td>
<td>int main(/<em>...</em>/) {</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>// ... initialization ...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>qsort(/<em>...</em>/);</td>
<td>16.3</td>
<td>0.0</td>
</tr>
<tr>
<td>12</td>
<td>// ... use array ...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>}</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Case study: qsort

Invocation trees

Cilkprof interprets the program execution in terms of its invocation tree.

Executed call site (with line number)

Function instantiation

Parallel call sites
Case study: qsort

Separately profiling work and span

Cilkprof breaks down both the work and the span of the program.

- Instantiations **on the work** are all instantiations in the computation.
- Instantiations **on the span** are the instantiations on the critical path.
Case study: qsort

Handling multiple executions of a call site

How does Cilkprof aggregate measurements from multiple executions of the same call site?

**Simple idea:** Just add them.

**Problem:** This strategy overcounts measurements for recursive functions.
How does Cilkprof aggregate measurements from multiple executions of the same call site?

**Simple idea:** Just add them.

**Problem:** This strategy overcounts measurements for recursive functions.
How does Cilkprof aggregate measurements from multiple executions of the same call site?

**Simple idea:** Just add them.

**Problem:** This strategy overcounts measurements for recursive functions.
Handling multiple executions of a call site: top-caller

Idea: Selectively measure “top-caller” executions of each call site.

A call site $s$ belongs to a particular caller function $F$.

A top-caller execution of $s$ has only one instantiation of $F$ above it.
Handling multiple executions of a call site: top-caller

Idea: Selectively measure “top-caller” executions of each call site.

A call site \( s \) belongs to a particular caller function \( F \).

A top-caller execution of \( s \) has only one instantiation of \( F \) above it.
Handling multiple executions of a call site: top-caller

Idea: Selectively measure “top-caller” executions of each call site.

A call site \( s \) belongs to a particular caller function \( F \).

A top-caller execution of \( s \) has only one instantiation of \( F \) above it.
Handling multiple executions of a call site: local

Idea: Measure every execution of each call site, but only record the **local** computation of the instantiations.

- Exclude the computation performed by child instantiations.

This is similar to gprof’s “self time.”
Handling multiple executions of a call site: local

Idea: Measure every execution of each call site, but only record the local computation of the instantiations.

- Exclude the computation performed by child instantiations.

This is similar to gprof’s “self time.”
Case study: qsort

Handling multiple executions of a call site: local

Idea: Measure every execution of each call site, but only record the local computation of the instantiations.

- Exclude the computation performed by child instantiations.

This is similar to gprof’s “self time.”
## Finding the scalability bottleneck in qsort

<table>
<thead>
<tr>
<th></th>
<th>On work (gigacycles)</th>
<th>On span (gigacycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Top-caller work</td>
<td>Local work</td>
</tr>
<tr>
<td></td>
<td>Top-caller span</td>
<td>Local span</td>
</tr>
<tr>
<td>01 void qsort(/<em>...</em>/) {</td>
<td></td>
<td></td>
</tr>
<tr>
<td>02 /* ... base case ... */</td>
<td></td>
<td></td>
</tr>
<tr>
<td>03 part = partition(/<em>...</em>/);</td>
<td>0.6</td>
<td>10.4</td>
</tr>
<tr>
<td>04 cilk_spawn qsort(/<em>...</em>/);</td>
<td>1.5</td>
<td>3.2</td>
</tr>
<tr>
<td>05 qsort(/<em>...</em>/);</td>
<td>14.3</td>
<td>2.7</td>
</tr>
<tr>
<td>06 cilk_sync;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>07 }</td>
<td></td>
<td></td>
</tr>
<tr>
<td>08</td>
<td></td>
<td></td>
</tr>
<tr>
<td>09 int main(/<em>...</em>/) {</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10 // ... initialization ...</td>
<td>16.3</td>
<td>0.0</td>
</tr>
<tr>
<td>11 qsort(/<em>...</em>/);</td>
<td></td>
<td>3.3</td>
</tr>
<tr>
<td>12 // ... use array ...</td>
<td></td>
<td>0.0</td>
</tr>
<tr>
<td>13 }</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Finding the scalability bottleneck in qsort

**Property:** For the topmost `qsort` instantiation, its top-caller work equals its local work plus the total top-caller work of its children.

<table>
<thead>
<tr>
<th>On work (gigacycles)</th>
<th>On span (gigacycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Top-caller work</td>
<td>Local work</td>
</tr>
<tr>
<td>Top-caller span</td>
<td>Local span</td>
</tr>
</tbody>
</table>

|  |  |  |  |  |
|---|---|---|---|
| 0.6 | 10.4 | 0.6 | 3.3 |
| 1.5 | 3.2 | 0.0 | 0.0 |
| 14.3 | 2.7 | 2.8 | 0.0 |

```c
01 void qsort(/*...*/) {
02    /* ... base case ... */
03    part = partition(/*...*/);
04    cilk_spawn qsort(/*...*/);
05    qsort(/*...*/);
06    cilk_sync;
07 }
08
09 int main(/*...*/) {
10    // ... initialization ... 
11    qsort(/*...*/);
12    // ... use array ... 
13 }
```

Schardl et al. The Cilkprof Scalability Profiler SPAA 2015 22 / 34
## Case study: `qsort`

### Finding the scalability bottleneck in `qsort`

**Property:** For the topmost `qsort` instantiation, its **top-caller work** equals its **local work** plus the total **local work of its children**.

<table>
<thead>
<tr>
<th></th>
<th>On work (gigacycles)</th>
<th>On span (gigacycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Top-caller work</td>
<td>Local work</td>
</tr>
<tr>
<td>Top-caller</td>
<td>0.6</td>
<td>10.4</td>
</tr>
<tr>
<td>Local work</td>
<td>1.5</td>
<td>3.2</td>
</tr>
<tr>
<td>Top-caller span</td>
<td>14.3</td>
<td>2.7</td>
</tr>
</tbody>
</table>

```c
01 void qsort(/*...*/) {
    /* ... base case ... */
03 part = partition(/*...*/);
04 cilk_spawn qsort(/*...*/);
05 qsort(/*...*/);
06 cilk_sync;
07 }

09 int main(/*...*/) {
    // ... initialization ... 
11 qsort(/*...*/); 
12 // ... use array ... 
13 }
```
Finding the scalability bottleneck in qsort

<table>
<thead>
<tr>
<th></th>
<th>On work (gigacycles)</th>
<th>On span (gigacycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Top-caller work</td>
<td>Local work</td>
</tr>
<tr>
<td></td>
<td>Top-caller span</td>
<td>Local span</td>
</tr>
<tr>
<td>part = partition(/<em>...</em>/);</td>
<td>0.6</td>
<td>10.4</td>
</tr>
<tr>
<td>cilk_spawn qsort(/<em>...</em>/);</td>
<td>1.5</td>
<td>3.2</td>
</tr>
<tr>
<td>qsort(/<em>...</em>/);</td>
<td>14.3</td>
<td>2.7</td>
</tr>
<tr>
<td>cilk_sync;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

void qsort(/*...*/) {
    /* ... base case ... */
    part = partition(/*...*/);
    cilk_spawn qsort(/*...*/);
    qsort(/*...*/);
    cilk_sync;
}

int main(/*...*/) {
    // ... initialization ...
    qsort(/*...*/);
    // ... use array ...
}
Finding the scalability bottleneck in qsort

**Property:** For the topmost qsort instantiation, its **top-caller span** equals its local span plus the total top-caller span of its children.

<table>
<thead>
<tr>
<th></th>
<th>On work (gigacycles)</th>
<th>On span (gigacycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Top-caller work</td>
<td>Local work</td>
</tr>
<tr>
<td></td>
<td>Top-caller span</td>
<td>Local span</td>
</tr>
<tr>
<td>part = partition()</td>
<td>0.6</td>
<td>10.4</td>
</tr>
<tr>
<td>cilk_spawn qsort()</td>
<td>1.5</td>
<td>3.2</td>
</tr>
<tr>
<td>qsort()</td>
<td>14.3</td>
<td>2.7</td>
</tr>
<tr>
<td>}</td>
<td>0.6</td>
<td>3.3</td>
</tr>
<tr>
<td>int main()</td>
<td>16.3</td>
<td>0.0</td>
</tr>
<tr>
<td>qsort()</td>
<td>3.3</td>
<td>0.0</td>
</tr>
<tr>
<td>}</td>
<td>0.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

```c
01 void qsort(/*...*/ ) {
02     /* ... base case ... */
03     part = partition(/*...*/);
04     cilk_spawn qsort(/*...*/);
05     qsort(/*...*/);
06     cilk_sync;
07 }
08
09 int main(/*...*/ ) {
10     // ... initialization ...
11     qsort(/*...*/);
12     // ... use array ...
13 }
```
Finding the scalability bottleneck in qsort

**Property:** For the topmost `qsort` instantiation, its top-caller span equals its local span plus the total local span of its children.

<table>
<thead>
<tr>
<th>On work (gigacycles)</th>
<th>On span (gigacycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Top-caller work</td>
<td>Local work</td>
</tr>
<tr>
<td>Top-caller span</td>
<td>Local span</td>
</tr>
</tbody>
</table>

```c
01 void qsort(/*...*/) {
02 /* ... base case ... */
03 part = partition(/*...*/);
04 cilk_spawn qsort(/*...*/);
05 qsort(/*...*/);
06 cilk_sync;
07 }

08

09 int main(/*...*/) {
10 // ... initialization ...
11 qsort(/*...*/);
12 // ... use array ...
13 }
```

Schardl et al. The Cilkprof Scalability Profiler SPAA 2015 22 / 34
Outline

1. Case study: qsort
2. Case study: pbfs
3. Profiling the work and span
4. Conclusion
We were stumped for months trying to pinpoint a scalability bottleneck in pbfs, a Cilk program to perform parallel breadth-first search.

A back-of-the-envelope calculation suggests that pbfs can achieve parallelism of 200–400.

Cilkview measured the parallelism of pbfs to be only 12.
Solved in two hours with Cilkprof

Using a prototype of Cilkprof, we were able to pinpoint and fix the scalability bottleneck in *pbfs* in 2 hours.

- Sorting Cilkprof’s output revealed three main contributors to the span:
  - `parseBinaryFile()`: Routine to read the input graph.
  - `Graph()`: Constructor for the graph data structure.
  - `pbfs_proc_Node()`: Base case of the primary recursive routine.
- We found and fixed a mistuned constant in `pbfs_proc_Node()`, improving the parallelism of *pbfs* by a factor of 5.
Outline

1. Case study: qsort
2. Case study: pbfs
3. Profiling the work and span
4. Conclusion
Computing work and span: variables

Cilkprof augments an algorithm that incrementally computes the work and span of a whole program execution.

For each instantiation $F$:

<table>
<thead>
<tr>
<th>Variables</th>
<th>Work</th>
<th>Span</th>
</tr>
</thead>
<tbody>
<tr>
<td>$F. w$</td>
<td>$F. p$</td>
<td>$F. \ell$</td>
</tr>
<tr>
<td>$F. c$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Invariant:** When $F$ returns:
- $F. w$ stores its work.
- $F. p$ stores its span.
Profiling the work and span

Computing work and span: variables

Cilkprof augments an algorithm that incrementally computes the work and span of a whole program execution.

For each instantiation \( F \):

<table>
<thead>
<tr>
<th>Variables</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Work</strong></td>
<td>( F.w )</td>
</tr>
<tr>
<td>( F.p )</td>
<td></td>
</tr>
<tr>
<td><strong>Span</strong></td>
<td>( F.\ell )</td>
</tr>
<tr>
<td>( F.c )</td>
<td></td>
</tr>
</tbody>
</table>

**Invariant:** When \( F \) returns:
- \( F.w \) stores its work.
- \( F.p \) stores its span.

**Corollary:** When \( \text{main} \) returns:
- \( \text{main}.w \) stores the computation’s work.
- \( \text{main}.p \) stores the computation’s span.
Computing work and span: sum and max

Cilkprof computes work and span incrementally by taking sums and (something like) maxes of the work and span variables.

<table>
<thead>
<tr>
<th>F spawns or calls G:</th>
<th>Called G returns to F:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 let G.w = 0</td>
<td>5 G.p += G.c</td>
</tr>
<tr>
<td>2 let G.p = 0</td>
<td>6 F.w += G.w</td>
</tr>
<tr>
<td>3 let G.ℓ = 0</td>
<td>7 F.c += G.p</td>
</tr>
<tr>
<td>4 let G.c = 0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Spawned G returns to F:</th>
<th>F syncs:</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 G.p += G.c</td>
<td>14 if F.c &gt; F.ℓ</td>
</tr>
<tr>
<td>9 F.w += G.w</td>
<td>15 F.p += F.c</td>
</tr>
<tr>
<td>10 if F.c + G.p &gt; F.ℓ</td>
<td>16 else</td>
</tr>
<tr>
<td>11 F.ℓ = G.p</td>
<td>17 F.p += F.ℓ</td>
</tr>
<tr>
<td>12 F.p += F.c</td>
<td>18 F.c = 0</td>
</tr>
<tr>
<td>13 F.c = 0</td>
<td>19 F.ℓ = 0</td>
</tr>
</tbody>
</table>

This pseudocode performs the following types of operations:

- Initialization
- Sum
- Max (or something like it)
Computing work and span: sum and max

Cilkprof computes work and span incrementally by taking sums and (something like) maxes of the work and span variables.

**F** spawns or calls **G**:  
1. let **G.**w = 0  
2. let **G.**p = 0  
3. let **G.**l = 0  
4. let **G.**c = 0

**Called G** returns to **F**:  
5. **G.**p += **G.**c  
6. **F.**w += **G.**w  
7. **F.**c += **G.**p

**Spawned G** returns to **F**:  
8. **G.**p += **G.**c  
9. **F.**w += **G.**w  
10. if **F.**c + **G.**p > **F.**l  
11. **F.**l = **G.**p  
12. **F.**p += **F.**c  
13. **F.**c = 0

**F** syncs:  
14. if **F.**c > **F.**l  
15. **F.**p += **F.**c  
16. else  
17. **F.**p += **F.**l  
18. **F.**c = 0  
19. **F.**l = 0

This pseudocode performs the following types of operations:  
- Initialization  
- Sum  
- Max (or something like it)
Computing work and span: sum and max

Cilkprof computes work and span incrementally by taking sums and (something like) maxes of the work and span variables.

<table>
<thead>
<tr>
<th>F spawns or calls G:</th>
<th>Called G returns to F:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 let G.w = 0</td>
<td>5 G.p += G.c</td>
</tr>
<tr>
<td>2 let G.p = 0</td>
<td>6 F.w += G.w</td>
</tr>
<tr>
<td>3 let G.ℓ = 0</td>
<td>7 F.c += G.p</td>
</tr>
<tr>
<td>4 let G.c = 0</td>
<td></td>
</tr>
</tbody>
</table>

Spawned G returns to F:

| 8 G.p += G.c        | 14 if F.c > F.ℓ       |
| 9 F.w += G.w       | 15 F.p += F.c         |
| 10 if F.c + G.p > F.ℓ | 16 else               |
| 11 F.ℓ = G.p       | 17 F.p += F.ℓ         |
| 12 F.p += F.c      | 18 F.c = 0            |
| 13 F.c = 0         | 19 F.ℓ = 0            |

This pseudocode performs the following types of operations:

- Initialization
- Sum
- Max (or something like it)
Computing work and span: sum and max

Cilkprof computes work and span incrementally by taking sums and (something like) maxes of the work and span variables.

<table>
<thead>
<tr>
<th>$F$ spawns or calls $G$:</th>
<th>Called $G$ returns to $F$:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 let $G.w = 0$</td>
<td>5 $G.p += G.c$</td>
</tr>
<tr>
<td>2 let $G.p = 0$</td>
<td>6 $F.w += G.w$</td>
</tr>
<tr>
<td>3 let $G.\ell = 0$</td>
<td>7 $F.c += G.p$</td>
</tr>
<tr>
<td>4 let $G.c = 0$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Spawned $G$ returns to $F$:</th>
<th>$F$ syncs:</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 $G.p += G.c$</td>
<td>14 if $F.c &gt; F.\ell$</td>
</tr>
<tr>
<td>9 $F.w += G.w$</td>
<td>15 $F.p += F.c$</td>
</tr>
<tr>
<td>10 if $F.c + G.p &gt; F.\ell$</td>
<td>16 else</td>
</tr>
<tr>
<td>11 $F.\ell = G.p$</td>
<td>17 $F.p += F.\ell$</td>
</tr>
<tr>
<td>12 $F.p += F.c$</td>
<td>18 $F.c = 0$</td>
</tr>
<tr>
<td>13 $F.c = 0$</td>
<td>19 $F.\ell = 0$</td>
</tr>
</tbody>
</table>

This pseudocode performs the following types of operations:

- Initialization
- Sum
- Max (or something like it)
Computing work and span profiles

**Key Idea:** Attach a profile to each work and span variable.

For each instantiation $F$:

**Variables**

<table>
<thead>
<tr>
<th>Work</th>
<th>$F.w$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$F.p$</td>
<td></td>
</tr>
<tr>
<td>Span</td>
<td>$F.\ell$</td>
</tr>
<tr>
<td>$F.c$</td>
<td></td>
</tr>
</tbody>
</table>

**Invariant:** When $F$ returns:
- $F.w$ stores its work.
- $F.p$ stores its span.

**Corollary:** When `main` returns:
- `main.w` stores the computation's work.
- `main.p` stores the computation's span.
Computing work and span profiles

**Key Idea:** Attach a profile to each work and span variable.

For each instantiation $F$:

<table>
<thead>
<tr>
<th>Variables</th>
<th>Profiles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Work</td>
<td>$F.w$</td>
</tr>
<tr>
<td></td>
<td>$F.p$</td>
</tr>
<tr>
<td>Span</td>
<td>$F.\ell$</td>
</tr>
<tr>
<td></td>
<td>$F.c$</td>
</tr>
</tbody>
</table>

**Invariant:** When $F$ returns:
- $F.w$ stores its work.
- $F.p$ stores its span.

**Corollary:** When `main` returns:
- `main.w` stores the computation’s work.
- `main.p` stores the computation’s span.
Computing work and span profiles

**Key Idea:** Attach a profile to each work and span variable.

For each instantiation \( F \):

<table>
<thead>
<tr>
<th>Variables</th>
<th>Profiles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Work</td>
<td></td>
</tr>
<tr>
<td>( F.w )</td>
<td>( F.w.prof )</td>
</tr>
<tr>
<td>( F.p )</td>
<td>( F.p.prof )</td>
</tr>
<tr>
<td>( F.\ell )</td>
<td>( F.p.prof )</td>
</tr>
<tr>
<td>( F.c )</td>
<td>( F.c.prof )</td>
</tr>
<tr>
<td>Span</td>
<td></td>
</tr>
</tbody>
</table>

**Invariant:** When \( F \) returns:
- \( F.w \) stores its work.
- \( F.p \) stores its span.
- \( F.w.prof \) stores the *profile* of \( F \)’s work.
- \( F.p.prof \) stores the *profile* of \( F \)’s span.

**Corollary:** When \( \text{main} \) returns:
- \( \text{main}.w \) stores the computation’s work.
- \( \text{main}.p \) stores the computation’s span.
# Computing work and span profiles

**Key Idea:** Attach a profile to each work and span variable.

For each instantiation $F$:

<table>
<thead>
<tr>
<th>Variables</th>
<th>Profiles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Work</td>
<td></td>
</tr>
<tr>
<td>$F.w$</td>
<td>$F.w.prof$</td>
</tr>
<tr>
<td>$F.p$</td>
<td>$F.p.prof$</td>
</tr>
<tr>
<td>$F.\ell$</td>
<td>$F.p.prof$</td>
</tr>
<tr>
<td>$F.c$</td>
<td>$F.c.prof$</td>
</tr>
<tr>
<td>Span</td>
<td></td>
</tr>
</tbody>
</table>

**Invariant:** When $F$ returns:
- $F.w$ stores its work.
- $F.p$ stores its span.
- $F.w.prof$ stores the profile of $F$’s work.
- $F.p.prof$ stores the profile of $F$’s span.

**Corollary:** When main returns:
- main.$w$ stores the computation’s work.
- main.$p$ stores the computation’s span.
- main.$w.prof$ stores the profile of the computation on the work.
- main.$p.prof$ stores the profile of the computation on the span.
Profiling the work and span

Maintaining a profile

The prof data structure supports the following operations:

- **INIT()**: Initialize a prof $R$ to be empty.
- **ASSIGN($R, R'$)**: Replace the contents of prof $R$ with that of prof $R'$, then discard the contents of $R'$.
- **UNION($R, R'$)**: Update the prof $R$ element-wise with the contents of the $R'$, then discard the contents of $R'$.
- **UPDATE($R, \langle s, v \rangle$)**: If no record $v'$ associated with call site $s$ already exists in $R$, store $\langle s, v \rangle$ into $R$. Otherwise, store $\langle s, v' + v \rangle$. 

Schardl et al. The Cilkprof Scalability Profiler SPAA 2015 28 / 34
Computing work and span profiles: **UNION and ASSIGN**

Called $G$ returns to $F$:

1. \( G.p \mathrel{+}= G.c \)
2. \( F.w \mathrel{+}= G.w \)
3. \( F.c \mathrel{+}= G.p \)

Spawned $G$ returns to $F$:

4. \( F.c + G.p > F.\ell \)
5. \( F.\ell = G.p \)
6. \( F.p \mathrel{+}= F.c \)
7. \( F.c = 0 \)

When variables are **summed**, Cilkprof uses **UNION** to combine their profiles.

When variables are **maxed**, Cilkprof uses **ASSIGN** to discard the profile of the smaller variable.

- **UNION** is also used when the max-like pseudocode adds variables.
Computing work and span profiles: **UNION** and **ASSIGN**

Called $G$ returns to $F$:

1. $G.p += G.c$
2. **UNION**(G.p.prof, G.c.prof)
3. $F.w += G.w$
4. **UNION**(F.w.prof, G.w.prof)
5. $F.c += G.p$
6. **UNION**(F.c.prof, G.p.prof)

Spawned $G$ returns to $F$:

7. **if** $F.c + G.p > F.\ell$
   8. $F.\ell = G.p$
9. $F.p += F.c$
10. $F.c = 0$

When variables are summed, Cilkprof uses **UNION** to combine their profiles.

When variables are maxed, Cilkprof uses **ASSIGN** to discard the profile of the smaller variable.

- **UNION** is also used when the max-like pseudocode adds variables.
Computing work and span profiles: UNION and ASSIGN

Called $G$ returns to $F$:

1. $G.p += G.c$
2. $\text{UNION}(G.p.prof, G.c.prof)$
3. $F.w += G.w$
4. $\text{UNION}(F.w.prof, G.w.prof)$
5. $F.c += G.p$
6. $\text{UNION}(F.c.prof, G.p.prof)$

Spawned $G$ returns to $F$:

7. if $F.c + G.p > F.\ell$
   8. $F.\ell = G.p$
   9. $\text{ASSIGN}(F.\ell.prof, G.p.prof)$
10. $F.p += F.c$
11. $\text{UNION}(F.p.prof, F.c.prof)$
12. $F.c = 0$

When variables are summed, Cilkprof uses UNION to combine their profiles.

When variables are maxed, Cilkprof uses ASSIGN to discard the profile of the smaller variable.

- UNION is also used when the max-like pseudocode adds variables.

When variables are summed, Cilkprof uses UNION to combine their profiles.

When variables are maxed, Cilkprof uses ASSIGN to discard the profile of the smaller variable.

- UNION is also used when the max-like pseudocode adds variables.
Theorem: If each operation on a prof data structure takes $\Theta(1)$ time, then Cilkprof executes a given Cilk program with work $T_1$ in $\Theta(T_1)$ total time.
An intuitive `prof` data structure

Intuitively, a `prof` is just a hashtable mapping call sites to work and span values.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>(20, 9)</td>
</tr>
<tr>
<td>C</td>
<td>(6, 6)</td>
</tr>
<tr>
<td>E</td>
<td>(2, 2)</td>
</tr>
<tr>
<td>D</td>
<td>(10, 7)</td>
</tr>
<tr>
<td>B</td>
<td>(2, 2)</td>
</tr>
</tbody>
</table>

**Problem:** Combining two hashtables must be done element-wise, which takes linear time. This data structure increases Cilkprof’s overhead to $\Theta(S)$ in the worst case.
An intuitive *prof* data structure

Intuitively, a *prof* is just a hashtable mapping call sites to work and span values.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>(20,9)</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>(6,6)</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>(2,2)</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>(10,7)</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>(2,2)</td>
<td></td>
</tr>
</tbody>
</table>

**Problem:** Combining two hashtables must be done element-wise, which takes linear time. This data structure increases Cilkprof’s overhead to $\Theta(S)$ in the worst case.

- Merging hashtables is wasteful when they don’t contain many entries.
An intuitive prof data structure

Intuitively, a prof is just a hashtable mapping call sites to work and span values.

Problem: Combining two hashtables must be done element-wise, which takes linear time. This data structure increases Cilkprof’s overhead to $\Theta(S)$ in the worst case.

- Merging hashtables is wasteful when they don’t contain many entries.
- If the table contains few entries, then it’s more efficient to simply log what gets added to the table in, e.g., a linked-list.
The actual **prof** data structure transforms between a linked-list and a hashtable.

- When the linked-list gets too large — $\Theta(S)$ entries — convert it to a hashtable.
- An amortization argument justifies that all operations on this **prof** data structure take $\Theta(1)$ amortized time.
Outline

1. Case study: qsort
2. Case study: pbfs
3. Profiling the work and span
4. Conclusion
Summary

Cilkprof is a scalability profiler for Cilk programs.

- We modified the Cilk Plus/LLVM compiler to instrument functions, spawns, and syncs in a Cilk program.
  Available from https://github.com/neboat/{llvm,clang}.

- We implemented Cilkprof as a library to link into an instrumented Cilk program.
  Available from https://github.com/neboat/cilktools.

- Running the instrumented program linked with Cilkprof produces a spreadsheet attributing portions of the program’s work and span to each call site.

- Cilkprof profiles a Cilk program with work $T_1$ in $\Theta(T_1)$ time.

- Cilkprof incurs a geometric-mean slowdown of $1.9 \times$ and a maximum slowdown of $7.4 \times$. 
$ clang++ -O3 -g qsort.cpp -o qsort -fcilkplus -ldl -fcilktool-instr-c -lcilkprof

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>mm</td>
<td>0.99</td>
</tr>
<tr>
<td>dedup</td>
<td>1.03</td>
</tr>
<tr>
<td>lu</td>
<td>1.04</td>
</tr>
<tr>
<td>strassen</td>
<td>1.06</td>
</tr>
<tr>
<td>heat</td>
<td>1.07</td>
</tr>
<tr>
<td>cilksort</td>
<td>1.08</td>
</tr>
<tr>
<td>pbfs</td>
<td>1.10</td>
</tr>
<tr>
<td>fft</td>
<td>1.15</td>
</tr>
<tr>
<td>quicksort</td>
<td>1.20</td>
</tr>
<tr>
<td>nqueens</td>
<td>1.27</td>
</tr>
<tr>
<td>ferret</td>
<td>2.04</td>
</tr>
<tr>
<td>leiserchess</td>
<td>3.72</td>
</tr>
<tr>
<td>collision</td>
<td>4.37</td>
</tr>
<tr>
<td>cholesky</td>
<td>4.54</td>
</tr>
<tr>
<td>hevc</td>
<td>6.25</td>
</tr>
<tr>
<td>fib</td>
<td>7.36</td>
</tr>
</tbody>
</table>