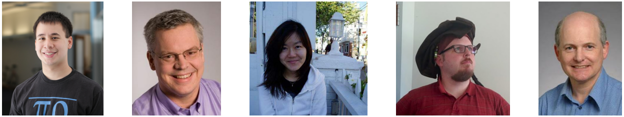
The Cilkprof Scalability Profiler



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:

```
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:

```
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;
}
```

Cilk parallel quicksort:

```
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;
}
```

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:

```
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;
}
```

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
```

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?

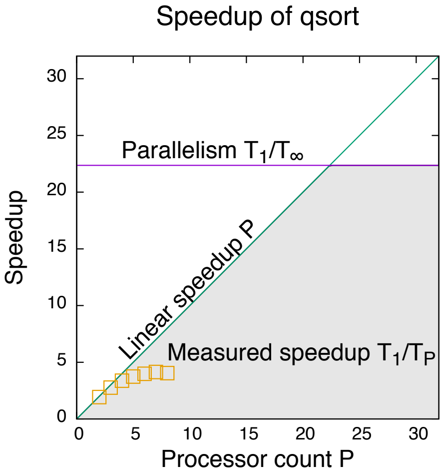
Prior art: Cilkview

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)



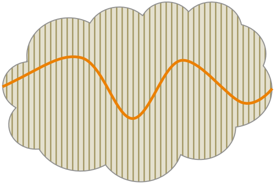
```
Work : 36,272,478,614 instructions
Span :  1,621,934,437 instructions
...
Parallelism : 22.36
...
```

Note: This output has been simplified for didactic purposes.

Work, span, and parallelism

Cilkview models the Cilk program performance as follows:

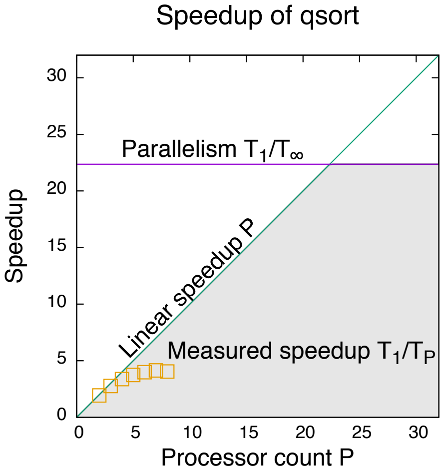
- Let T_P denote the execution time on P processors.
- ***Work*** is serial execution time T_1.
- ***Speedup*** on P processors is $T_1/T_P \leq P$.
  - ***Linear speedup*** occurs when $T_1/T_P = P$.
- ***Span*** is critical-path length T_∞.
- ***Parallelism*** is T_1/T_∞, the maximum possible speedup.



As a practical matter, a Cilk program should exhibit ***parallel slackness*** of 10 — it should exhibit $\geq 10P$ parallelism.

Work, span, and parallelism of `qsort`(100M)

Cilkview tells us that `qsort` does not exhibit much parallelism.



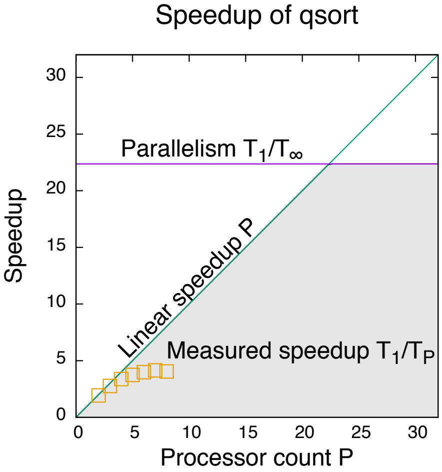
```
Work : 36,272,478,614 instructions
Span :  1,621,934,437 instructions
...
Parallelism : 22.36
...
```

Where is the bottleneck?

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

```
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;
}
```



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 `qsort`(100M)

Cilkprof gathers work and span measurements for every call site.

File	*Line*	*Top-caller work on work*	*Local work on work*	*Top-caller span on span*	*Local span on span*
`qsort.cpp`	5	0.6	10.4	0.6	3.3
`qsort.cpp`	6	1.5	3.2	0.0	0.0
`qsort.cpp`	7	14.3	2.7	0.0	0.0
`qsort.cpp`	13	16.3	0.0	3.3	0.0

Cilkprof's output for `qsort`(100M)

Cilkprof gathers work and span measurements for every call site.

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

int main(/*...*/) {
  // ... initialization ...
  qsort(/*...*/);
  // ... use array ...
}
```

	On work (gigacycles)		*On span (gigacycles)*	
	Top-caller work	*Local work*	*Top-caller span*	*Local span*
`part = partition(/*...*/);`	0.6	10.4	0.6	3.3
`cilk_spawn qsort(/*...*/);`	1.5	3.2	0.0	0.0
`qsort(/*...*/);`	14.3	2.7	2.8	0.0
`qsort(/*...*/);`	16.3	0.0	3.3	0.0

Compiler instrumentation

We used compiler instrumentation for its efficiency.

		Instrumentation	
	Sampling	*Binary*	*Compiler*
Example tools	`gprof`, `pprof`, `perf`	`valgrind`, `DynamoRio`, `Cilkview` (via `Pin`)	`gcov`, `tsan`, **`Cilkprof`**
Overhead	✓ None	× Lots	~ Some
Properties	✓ No recompilation necessary.	✓ No recompilation necessary.	~ Requires recompilation.
	× Statistical; don't know how to measure span via sampling.	~ Creates measurement error, but can still measure parallelism well enough.	~ Creates measurement error, but can still measure parallelism well enough.

Cilkprof's algorithm

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 ≈ 3000 call sites.

Cilkprof's empirical performance

Cilkprof is efficient in practice.

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

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`

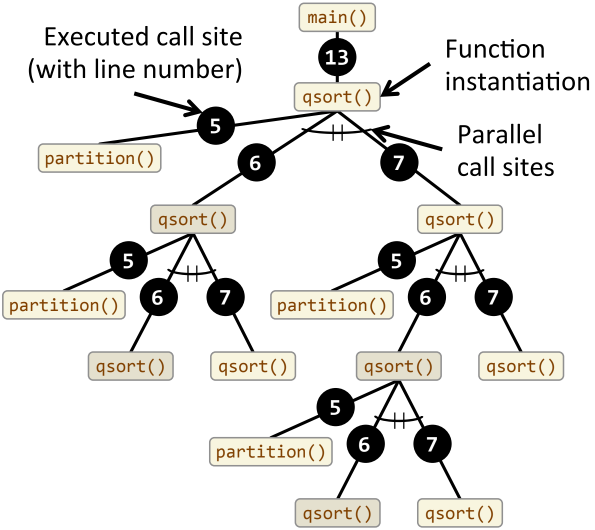
```
void qsort(/*...*/) {
  /* ... base case ... */
  part = partition(/*...*/);
  cilk_spawn qsort(/*...*/);
  qsort(/*...*/);
  cilk_sync;
}

int main(/*...*/) {
  // ... initialization ...
  qsort(/*...*/);
  // ... use array ...
}
```

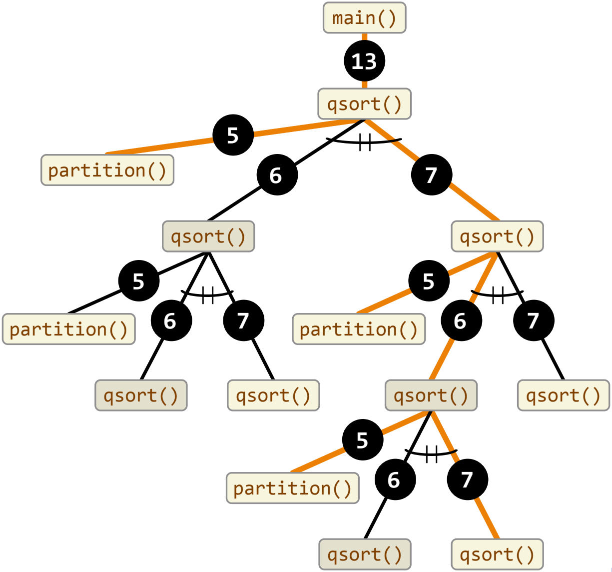
	On work (gigacycles)		*On span (gigacycles)*	
	Top-caller work	*Local work*	*Top-caller span*	*Local span*
`part = partition(/*...*/);`	0.6	10.4	0.6	3.3
`cilk_spawn qsort(/*...*/);`	1.5	3.2	0.0	0.0
`qsort(/*...*/);`	14.3	2.7	2.8	0.0
`qsort(/*...*/);`	16.3	0.0	3.3	0.0

Invocation trees

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



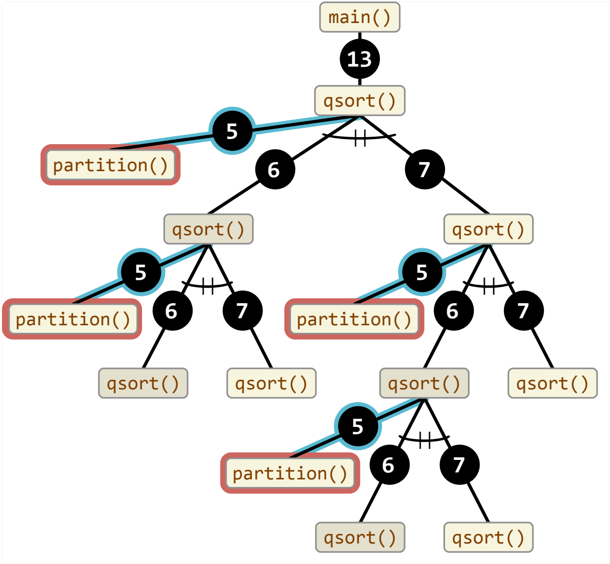
Separately profiling work and span



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.

Handling multiple executions of a call site

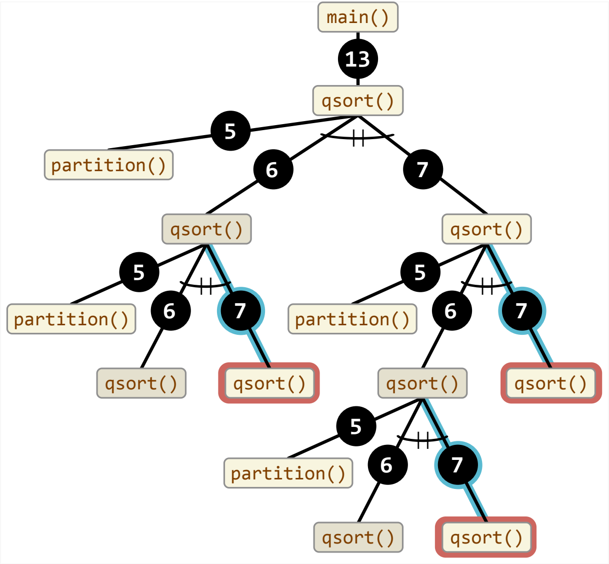


How does Cilkprof aggregate measurements from multiple executions of the same call site?

Simple idea: Just add them.

Problem: This strategy overcounts measurements for recursive functions.

Handling multiple executions of a call site

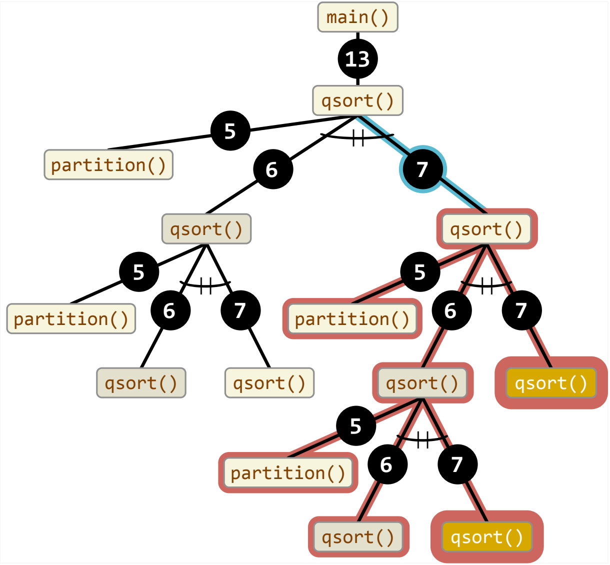


How does Cilkprof aggregate measurements from multiple executions of the same call site?

Simple idea: Just add them.

Problem: This strategy overcounts measurements for recursive functions.

Handling multiple executions of a call site

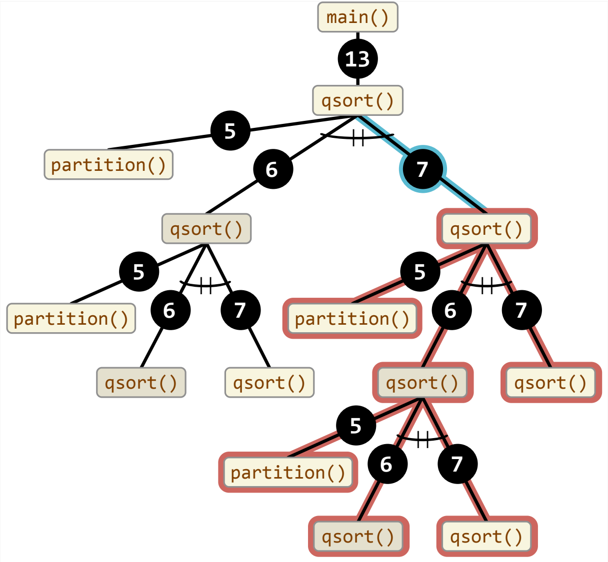


How does Cilkprof aggregate measurements from multiple executions of the same call site?

Simple idea: Just add them.

Problem: This strategy overcounts measurements for recursive functions.

Handling multiple executions of a call site: top-caller

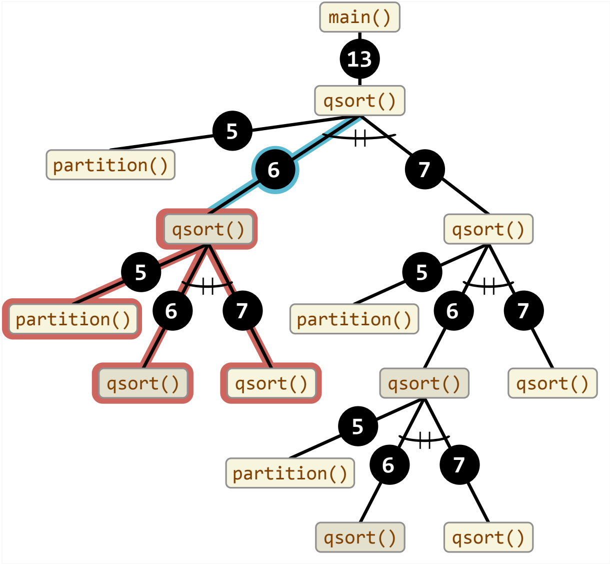


Idea: Selectively measure “top-caller” executions of each call site.

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

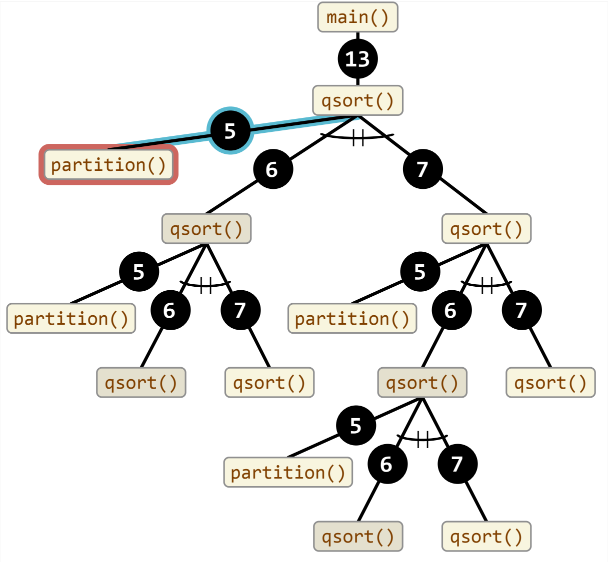


Idea: Selectively measure “top-caller” executions of each call site.

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

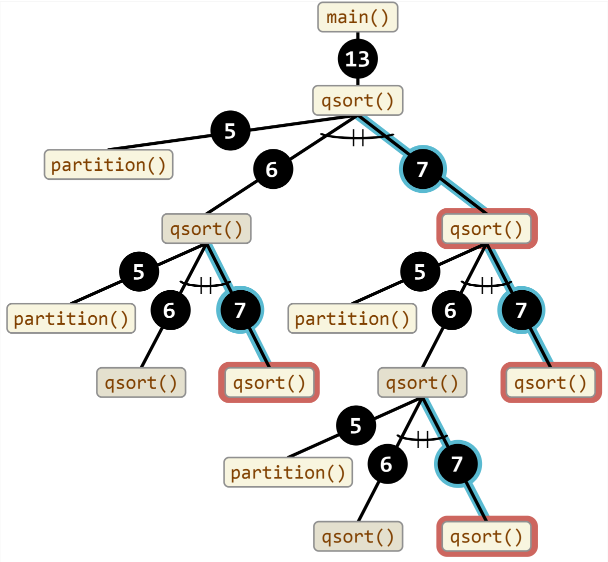


Idea: Selectively measure "top-caller" executions of each call site.

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

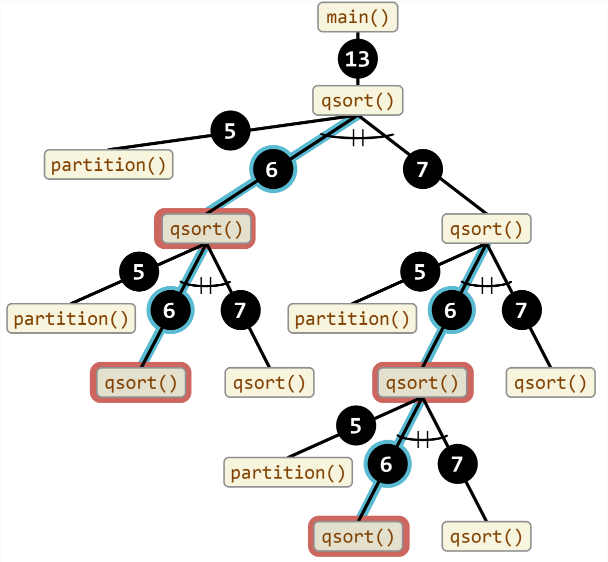


Idea: Measure every execution of each call site, but only record the ***local*** computation of the instantiations.

- Exclude the computation performed by child instantiations.

This is similar to `gprof`'s "self time."

Handling multiple executions of a call site: local

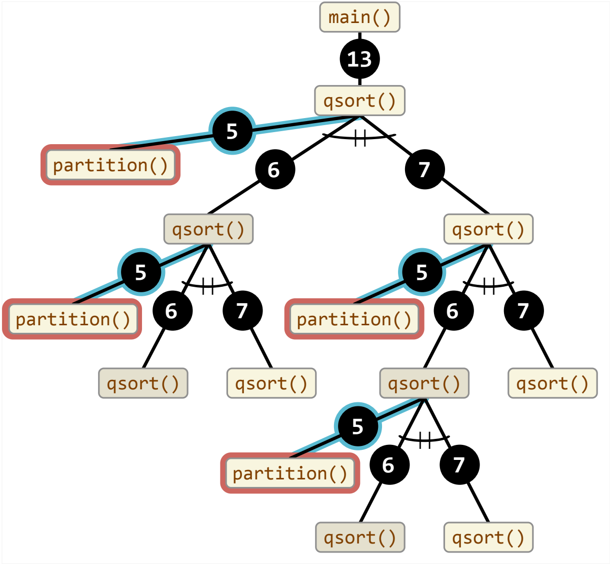


Idea: Measure every execution of each call site, but only record the ***local*** computation of the instantiations.

- Exclude the computation performed by child instantiations.

This is similar to `gprof`'s "self time."

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`

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

int main(/*...*/) {
  // ... initialization ...
  qsort(/*...*/);
  // ... use array ...
}
```

	On work (gigacycles)		*On span (gigacycles)*	
	Top-caller work	*Local work*	*Top-caller span*	*Local span*
`part = partition(/*...*/);`	0.6	10.4	0.6	3.3
`cilk_spawn qsort(/*...*/);`	1.5	3.2	0.0	0.0
`qsort(/*...*/);`	14.3	2.7	2.8	0.0
`qsort(/*...*/);`	16.3	0.0	3.3	0.0

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.

```
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 }
```

	On work (gigacycles)		*On span (gigacycles)*	
	Top-caller work	*Local work*	*Top-caller span*	*Local span*
03	0.6	10.4	0.6	3.3
04	1.5	3.2	0.0	0.0
05	14.3	2.7	2.8	0.0
11	16.3	0.0	3.3	0.0

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.

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

int main(/*...*/) {
  // ... initialization ...
  qsort(/*...*/);
  // ... use array ...
}
```

	On work (gigacycles)		*On span (gigacycles)*	
	Top-caller work	*Local work*	*Top-caller span*	*Local span*
`part = partition(/*...*/);`	0.6	10.4	0.6	3.3
`cilk_spawn qsort(/*...*/);`	1.5	3.2	0.0	0.0
`qsort(/*...*/);`	14.3	2.7	2.8	0.0
`qsort(/*...*/);`	16.3	0.0	3.3	0.0

Finding the scalability bottleneck in `qsort`

	On work (gigacycles)		*On span (gigacycles)*	
	Top-caller work	*Local work*	*Top-caller span*	*Local span*
`void qsort(/*...*/) {`				
`/* ... base case ... */`				
`part = partition(/*...*/);`	0.6	10.4	0.6	3.3
`cilk_spawn qsort(/*...*/);`	1.5	3.2	0.0	0.0
`qsort(/*...*/);`	14.3	2.7	2.8	0.0
`cilk_sync;`				
`}`				
`int main(/*...*/) {`				
`// ... initialization ...`				
`qsort(/*...*/);`	16.3	0.0	3.3	0.0
`// ... 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.

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

int main(/*...*/) {
  // ... initialization ...
  qsort(/*...*/);
  // ... use array ...
}
```

	On work (gigacycles)		*On span (gigacycles)*	
	Top-caller work	*Local work*	*Top-caller span*	*Local span*
`part = partition(/*...*/);`	0.6	10.4	0.6	3.3
`cilk_spawn qsort(/*...*/);`	1.5	3.2	0.0	0.0
`qsort(/*...*/);`	14.3	2.7	2.8	0.0
`qsort(/*...*/);`	16.3	0.0	3.3	0.0

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.

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

int main(/*...*/) {
  // ... initialization ...
  qsort(/*...*/);
  // ... use array ...
}
```

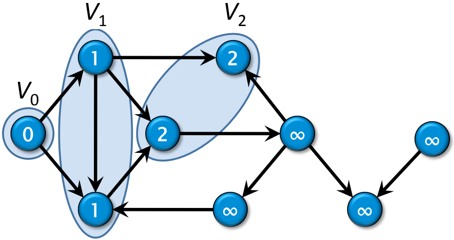
	On work (gigacycles)		*On span (gigacycles)*	
	Top-caller work	*Local work*	*Top-caller span*	*Local span*
`part = partition(/*...*/);`	0.6	10.4	0.6	3.3
`cilk_spawn qsort(/*...*/);`	1.5	3.2	0.0	0.0
`qsort(/*...*/);`	14.3	2.7	2.8	0.0
`qsort(/*...*/);`	16.3	0.0	3.3	0.0

Outline

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

Months of struggle with `pbfs`. . .

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:

	Variables
Work	$F.w$
Span	$F.p$ $F.\ell$ $F.c$

Invariant: When F returns:

- $F.w$ stores its work.
- $F.p$ stores its 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:

	Variables
Work	$F.w$
Span	$F.p$ $F.\ell$ $F.c$

Invariant: When F returns:

- $F.w$ stores its work.
- $F.p$ stores its span.

Corollary: When `main` returns:

- $\texttt{main}.w$ stores the computation's work.
- $\texttt{main}.p$ 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.

F spawns or calls G:

```
let $G.w = 0$
let $G.p = 0$
let $G.\ell = 0$
let $G.c = 0$
```

Called G returns to F:

```
$G.p \mathrel{+}= G.c$
$F.w \mathrel{+}= G.w$
$F.c \mathrel{+}= G.p$
```

Spawned G returns to F:

```
$G.p \mathrel{+}= G.c$
$F.w \mathrel{+}= G.w$
if $F.c + G.p > F.\ell$
    $F.\ell = G.p$
    $F.p \mathrel{+}= F.c$
    $F.c = 0$
```

F syncs:

```
if $F.c > F.\ell$
    $F.p \mathrel{+}= F.c$
else
    $F.p \mathrel{+}= F.\ell$
$F.c = 0$
$F.\ell = 0$
```

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.

F spawns or calls G:

```
let $G.w = 0$
let $G.p = 0$
let $G.\ell = 0$
let $G.c = 0$
```

Called G returns to F:

```
$G.p \mathrel{+}= G.c$
$F.w \mathrel{+}= G.w$
$F.c \mathrel{+}= G.p$
```

Spawned G returns to F:

```
$G.p \mathrel{+}= G.c$
$F.w \mathrel{+}= G.w$
if $F.c + G.p > F.\ell$
    $F.\ell = G.p$
    $F.p \mathrel{+}= F.c$
    $F.c = 0$
```

F syncs:

```
if $F.c > F.\ell$
    $F.p \mathrel{+}= F.c$
else
    $F.p \mathrel{+}= F.\ell$
$F.c = 0$
$F.\ell = 0$
```

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.

F spawns or calls G:

```
let $G.w = 0$
let $G.p = 0$
let $G.\ell = 0$
let $G.c = 0$
```

Called G returns to F:

```
$G.p \mathrel{+}= G.c$
$F.w \mathrel{+}= G.w$
$F.c \mathrel{+}= G.p$
```

Spawned G returns to F:

```
$G.p \mathrel{+}= G.c$
$F.w \mathrel{+}= G.w$
if $F.c + G.p > F.\ell$
    $F.\ell = G.p$
    $F.p \mathrel{+}= F.c$
    $F.c = 0$
```

F syncs:

```
if $F.c > F.\ell$
    $F.p \mathrel{+}= F.c$
else
    $F.p \mathrel{+}= F.\ell$
$F.c = 0$
$F.\ell = 0$
```

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.

F spawns or calls G:

```
let $G.w = 0$
let $G.p = 0$
let $G.\ell = 0$
let $G.c = 0$
```

Called G returns to F:

```
$G.p \mathrel{+}= G.c$
$F.w \mathrel{+}= G.w$
$F.c \mathrel{+}= G.p$
```

Spawned G returns to F:

```
$G.p \mathrel{+}= G.c$
$F.w \mathrel{+}= G.w$
if $F.c + G.p > F.\ell$
    $F.\ell = G.p$
    $F.p \mathrel{+}= F.c$
    $F.c = 0$
```

F syncs:

```
if $F.c > F.\ell$
    $F.p \mathrel{+}= F.c$
else
    $F.p \mathrel{+}= F.\ell$
$F.c = 0$
$F.\ell = 0$
```

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
Work	$F.w$
Span	$F.p$ $F.\ell$ $F.c$

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:

	Variables	*Profiles*
Work	$F.w$	$F.w.prof$
Span	$F.p$	$F.p.prof$
	$F.\ell$	$F.p.prof$
	$F.c$	$F.c.prof$

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:

	Variables	*Profiles*
Work	$F.w$	$F.w.prof$
Span	$F.p$	$F.p.prof$
	$F.\ell$	$F.p.prof$
	$F.c$	$F.c.prof$

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.

Computing work and span profiles

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

For each instantiation F:

	Variables	*Profiles*
Work	$F.w$	$F.w.prof$
Span	$F.p$	$F.p.prof$
	$F.\ell$	$F.p.prof$
	$F.c$	$F.c.prof$

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*.

Maintaining a profile

The `prof` data structure supports the following operations:

- $\textsc{Init}()$: Initialize a `prof` R to be empty.
- $\textsc{Assign}(R, R')$: Replace the contents of `prof` R with that of `prof` R', then discard the contents of R'.
- $\textsc{Union}(R, R')$: Update the `prof` R element-wise with the contents of the R', then discard the contents of R'.
- $\textsc{Update}(R, \langle s, v \rangle)$: If no record v' associated with call site s already exists in R, store $\langle s, v \rangle$ into R. Otherwise, store $\langle s, v' + v \rangle$.

Computing work and span profiles: UNION and ASSIGN

```
Called $G$ returns to $F$:
$G.p \mathrel{+}= G.c$

$F.w \mathrel{+}= G.w$

$F.c \mathrel{+}= G.p$

Spawned $G$ returns to $F$:
if $F.c + G.p > F.\ell$
    $F.\ell = G.p$

    $F.p \mathrel{+}= F.c$

    $F.c = 0$
```

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 $F$:
$G.p \mathrel{+}= G.c$
$\textsc{Union}(G.p.prof, G.c.prof)$
$F.w \mathrel{+}= G.w$
$\textsc{Union}(F.w.prof, G.w.prof)$
$F.c \mathrel{+}= G.p$
$\textsc{Union}(F.c.prof, G.p.prof)$
Spawned $G$ returns to $F$:
if $F.c + G.p > F.\ell$
    $F.\ell = G.p$

    $F.p \mathrel{+}= F.c$

    $F.c = 0$
```

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 $F$:
$G.p \mathrel{+}= G.c$
Union($G.p.prof, G.c.prof$)
$F.w \mathrel{+}= G.w$
Union($F.w.prof, G.w.prof$)
$F.c \mathrel{+}= G.p$
Union($F.c.prof, G.p.prof$)

Spawned $G$ returns to $F$:
if $F.c + G.p > F.\ell$
    $F.\ell = G.p$
    Assign($F.\ell.prof, G.p.prof$)
    $F.p \mathrel{+}= F.c$
    Union($F.p.prof, F.c.prof$)
    $F.c = 0$
```

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.

Cilkprof asymptotic running time

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.

A	(20,9)
C	(6,6)
E	(2,2)
D	(10,7)
B	(2,2)

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.

A	(20,9)
C	(6,6)
E	(2,2)
D	(10,7)
B	(2,2)

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.

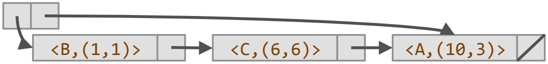
A	(20,9)
C	(6,6)
E	(2,2)
D	(10,7)
B	(2,2)

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

The `prof` data structure transforms between a linked-list and a hashtable.



A	(20,9)
C	(6,6)
E	(2,2)
D	(10,7)
B	(2,2)

- 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 $7.4\times$.

Thank you

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

Benchmark	*Overhead*
`mm`	0.99
`dedup`	1.03
`lu`	1.04
`strassen`	1.06
`heat`	1.07
`cilksort`	1.08
`pbfs`	1.10
`fft`	1.15
`quicksort`	1.20
`nqueens`	1.27
`ferret`	2.04
`leiserchess`	3.72
`collision`	4.37
`cholesky`	4.54
`hevc`	6.25
`fib`	7.36