Cilk Plus: Multicore extensions for C and C++

Matteo Frigo[1]

June 6, 2011

[1] Some slides courtesy of Prof. Charles E. Leiserson of MIT.

Intel® Cilk™ Plus

What is it?

C/C++ language extensions supporting fork/join and vector parallelism.

Features

- **Three simple keywords** for fork/join parallelism.
- **Cilkscreen**™ for accurate detection of determinacy races.
- **Cilkview**™ for analyzing parallelism.
- **Reducers** for resolving certain race conditions in a lock-free manner.
- Matlab™-style **array notation** for vector parallelism.
- Ships with the Intel® Parallel Building Blocks™.

Cilk language

Fibonacci

C++ elision

```
int fib(int n)
{
  if (n < 2) return n;
  else {
    int x, y;
    x = fib(n - 1);
    y = fib(n - 2);
    return x + y;
  }
}
```

Cilk

```
int fib(int n)
{
  if (n < 2) return n;
  else {
    int x, y;
    x = cilk_spawn fib(n - 1);
    y = fib(n - 2);
    cilk_sync;
    return x + y;
  }
}
```

Cilk is a **faithful** extension of C/C++. The **serial elision** of a Cilk program is a valid implementation.

Spawn and sync

Fibonacci

```
int fib(int n)
{
  if (n < 2) return n;
  else {
    int x, y;
    x = cilk_spawn fib(n - 1);
    y = fib(n - 2);
    cilk_sync;
    return x + y;
  }
}
```

cilk_spawn:

The child procedure may be executed in parallel with the parent.

`cilk_spawn` *grants permission* for parallel execution. It does not *command* parallel execution.

Spawn and sync

Fibonacci

```
int fib(int n)
{
  if (n < 2) return n;
  else {
    int x, y;
    x = cilk_spawn fib(n - 1);
    y = fib(n - 2);
    cilk_sync;
    return x + y;
  }
}
```

cilk_spawn:

The child procedure may be executed in parallel with the parent.

`cilk_spawn` *grants permission* for parallel execution. It does not *command* parallel execution.

cilk_sync:

Cannot be passed until all spawned children have returned.

Implicit `cilk_sync` at the end of every function.

Cactus stack

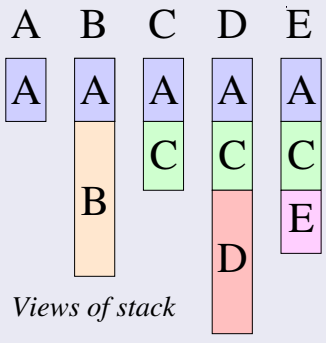
Cilk supports C's rule for pointers:

A pointer to stack space can be passed from parent to child, but not from child to parent. (Cilk also supports `malloc`.)

Cilk's cactus stack supports several stack views in parallel.

```
void A(void)
{
   cilk_spawn B();
   cilk_spawn C();
}

void C(void)
{
   cilk_spawn D();
   cilk_spawn E();
}
```



Parallel loops

Serial loop

```
for (int i = 0; i < n; ++i)
    foo(i);
```

Parallel loop

```
cilk_for (int i = 0; i < n; ++i)
    foo(i);
```

Serial loop that spawns

```
for (int i = 0; i < n; ++i)
    cilk_spawn foo(i);
cilk_sync;
```

cilk_for:

Executes all iterations in parallel. Implicit `cilk_sync` waits for all spawned iterations. Iterates over integers and random-access iterators.

Parallel loops

Serial loop

```
for (int i = 0; i < n; ++i)
    foo(i);
```

Parallel loop

```
cilk_for (int i = 0; i < n; ++i)
    foo(i);
```

Serial loop that spawns

```
for (int i = 0; i < n; ++i)
    cilk_spawn foo(i);
cilk_sync;
```

cilk_for:

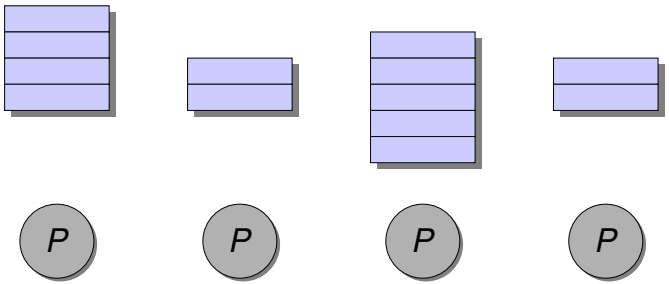
Executes all iterations in parallel. Implicit `cilk_sync` waits for all spawned iterations. Iterates over integers and random-access iterators.

Serial loop:

Long chain of dependencies: `++i` is serial.

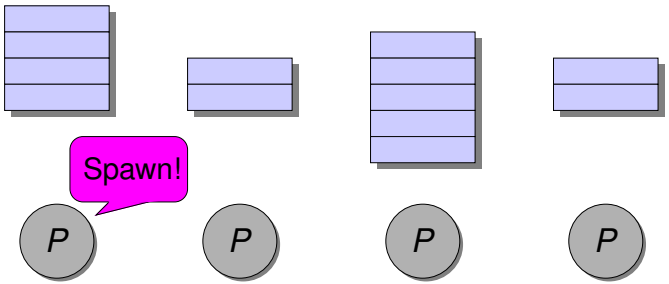
The Cilk work-stealing scheduler

Each worker maintains a work deque, and it manipulates the bottom of the deque like a stack.



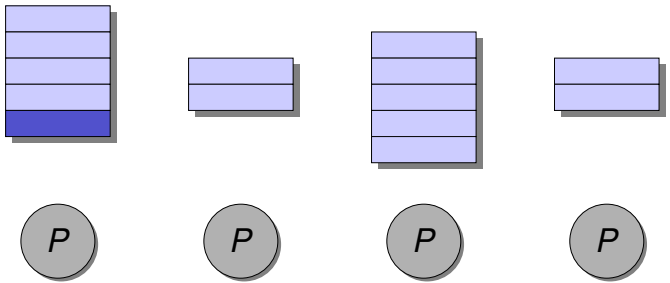
The Cilk work-stealing scheduler

Each worker maintains a work deque, and it manipulates the bottom of the deque like a stack.



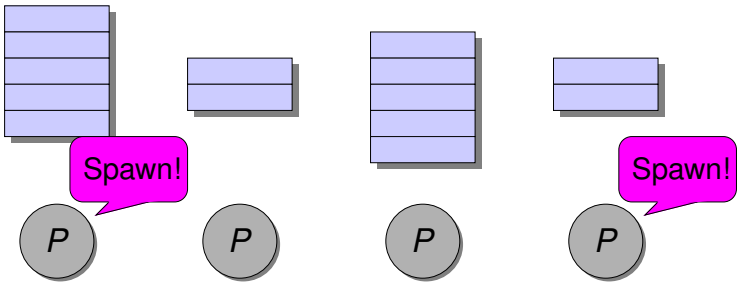
The Cilk work-stealing scheduler

Each worker maintains a work deque, and it manipulates the bottom of the deque like a stack.



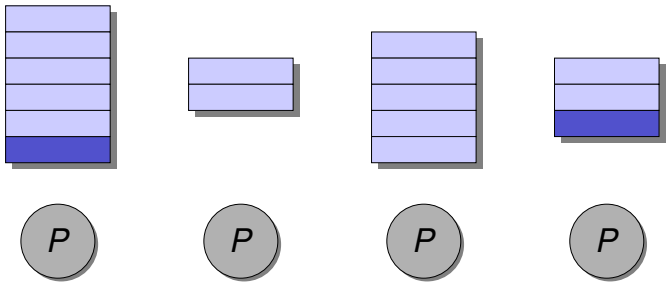
The Cilk work-stealing scheduler

Each worker maintains a work deque, and it manipulates the bottom of the deque like a stack.



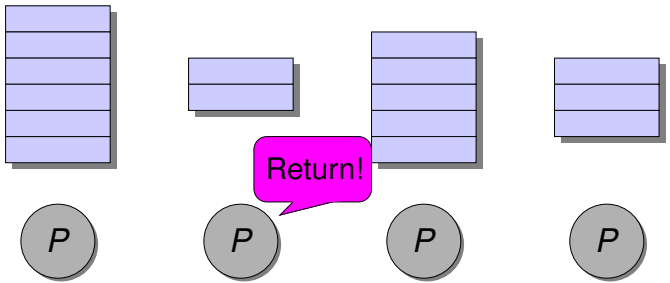
The Cilk work-stealing scheduler

Each worker maintains a work deque, and it manipulates the bottom of the deque like a stack.



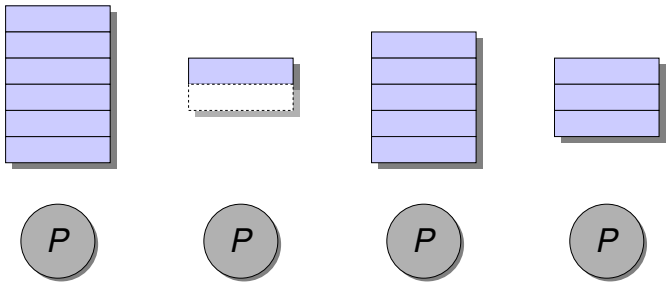
The Cilk work-stealing scheduler

Each worker maintains a work deque, and it manipulates the bottom of the deque like a stack.



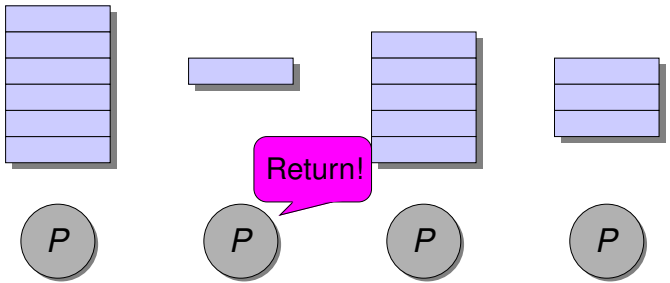
The Cilk work-stealing scheduler

Each worker maintains a work deque, and it manipulates the bottom of the deque like a stack.



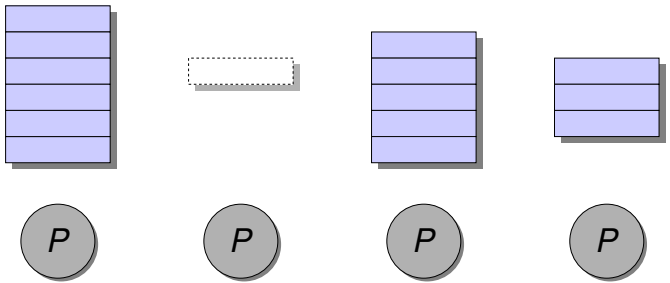
The Cilk work-stealing scheduler

Each worker maintains a work deque, and it manipulates the bottom of the deque like a stack.



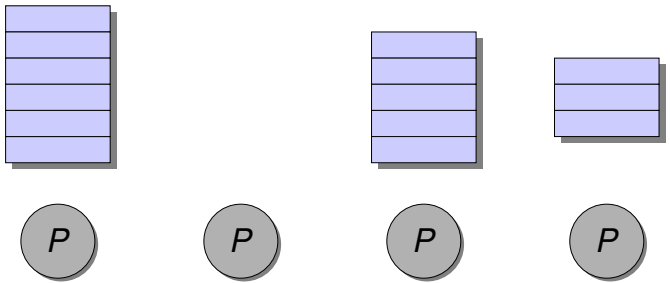
The Cilk work-stealing scheduler

Each worker maintains a work deque, and it manipulates the bottom of the deque like a stack.



The Cilk work-stealing scheduler

Each worker maintains a work deque, and it manipulates the bottom of the deque like a stack.

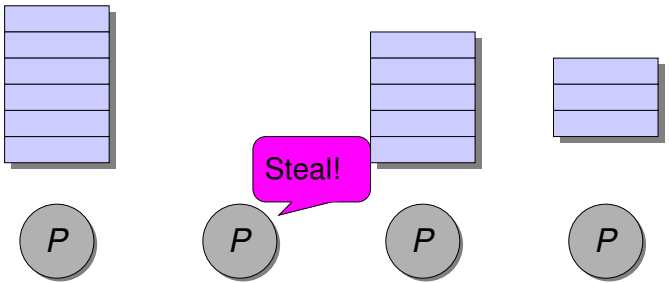


Randomized work stealing:

When a processor runs out of work, it steals a thread from the top of a random victim's deque.

The Cilk work-stealing scheduler

Each worker maintains a work deque, and it manipulates the bottom of the deque like a stack.

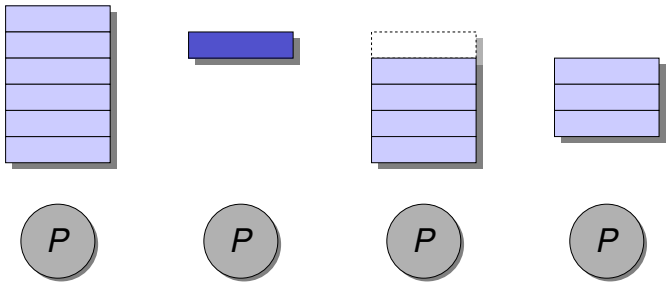


Randomized work stealing:

When a processor runs out of work, it steals a thread from the top of a random victim's deque.

The Cilk work-stealing scheduler

Each worker maintains a work deque, and it manipulates the bottom of the deque like a stack.

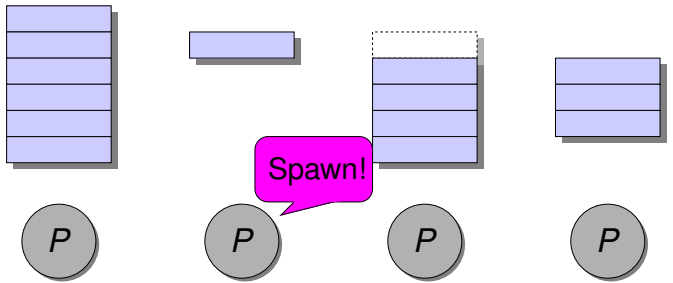


Randomized work stealing:

When a processor runs out of work, it steals a thread from the top of a random victim's deque.

The Cilk work-stealing scheduler

Each worker maintains a work deque, and it manipulates the bottom of the deque like a stack.

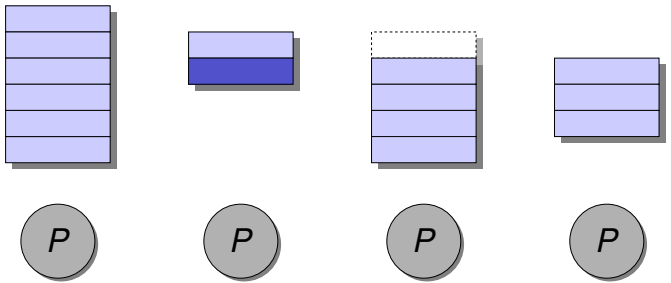


Randomized work stealing:

When a processor runs out of work, it steals a thread from the top of a random victim's deque.

The Cilk work-stealing scheduler

Each worker maintains a work deque, and it manipulates the bottom of the deque like a stack.



Randomized work stealing:

When a processor runs out of work, it steals a thread from the top of a random victim's deque.

The Cilkscreen race detector

Race bugs

Definition

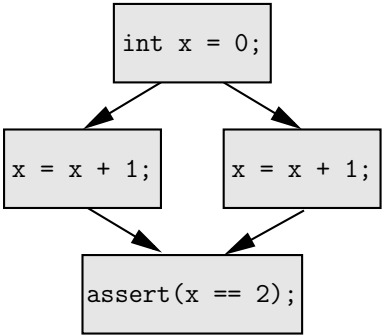
A **determinacy race** occurs when two logically parallel instructions access the same memory locations and at least one of the accesses is a write.

Example

```
int x = 0;

cilk_for (int i = 0; i < 2; ++i)
    x = x + 1;

assert(x == 2);
```



The Cilkscreen race detector

Correctness

Cilkscreen executes a program once on given input.

- If a race exists on a location, Cilkscreen reports a race on that location.
- No false positives: If a race does not exist, Cilkscreen reports no races.

Performance

- Constant memory overhead (about 4-5x), independent of the number of threads.
- (Almost) constant time overhead (about 10-50x), independent of the number of threads.

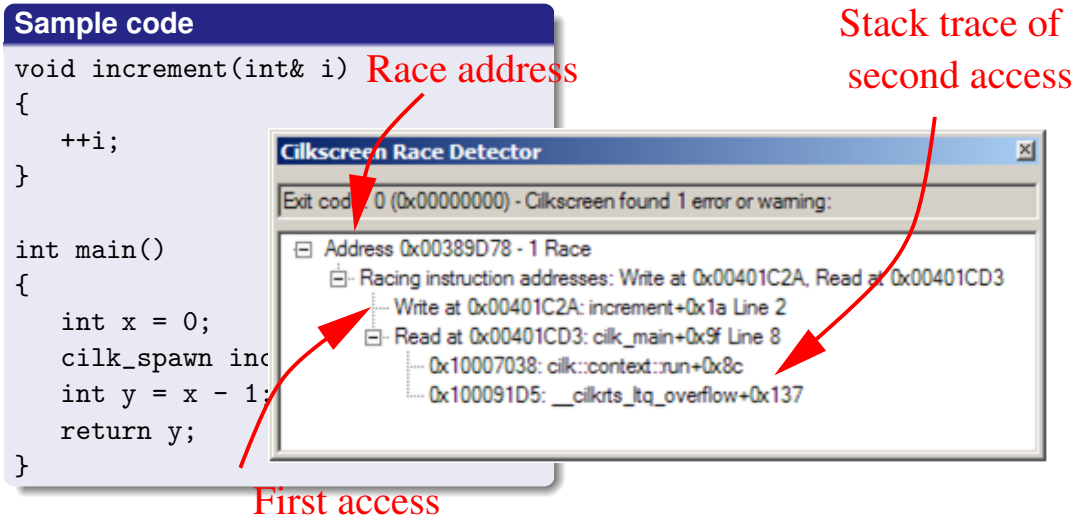
Cilkscreen screenshot

Sample code

```
void increment(int& i)
{
   ++i;
}

int main()
{
   int x = 0;
   cilk_spawn increment(x);
   int y = x - 1;
   return y;
}
```

Cilkscreen screenshot



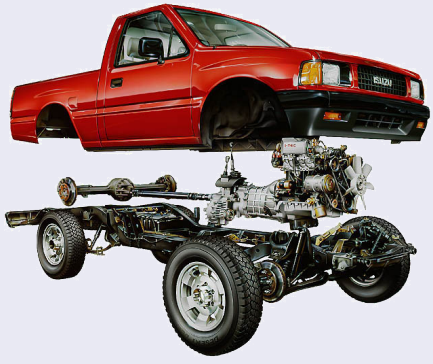
Cilkscreen in practice

- Executes production binaries. No separate “debug” binary is necessary.
- Identifies source locations and symbolic addresses involved in races.
- Reports the location of the first access and a stack trace of the second access.
- Is a debugger, not a symbolic theorem prover. It only analyzes paths that are actually executed.
- Understands locks.
- Supports various pragmas, e.g. for annotating intentional races.

Reducers

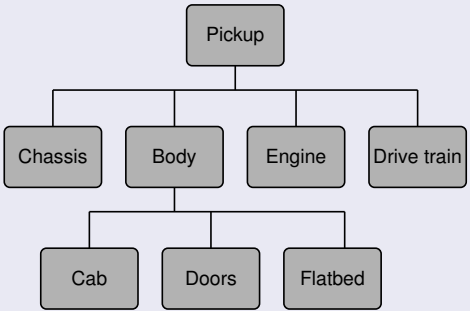
Collision detection

Mechanical assembly:



Internal representation:

Tree of subassemblies down to individual parts.



Problem:

Find all “collisions” between two assemblies.

Simplified collision detection

Goal:

Create a list of all the parts in a mechanical assembly that collide with a given target object.

Pseudo code:

```
Node *target;
std::list<Node *> output_list;

void walk(Node *x) {
  if (x->kind == Node::LEAF) {
    if (target->collides_with(x)) {
        output_list.push_back(x);
    }
  } else {
    for (Node::iterator child = x.begin();
         child != x.end();
         ++child) {
        walk(child);
    }
  }
}
```

Naive parallelization

Problem:

Race condition on the global variable `output_list`.

Pseudo code:

```
Node *target;
std::list<Node *> output_list;

void walk(Node *x) {
  if (x->kind == Node::LEAF) {
    if (target->collides_with(x)) {
        output_list.push_back(x);
    }
  } else {
    cilk_for (Node::iterator child = x.begin();
         child != x.end();
         ++child) {
        walk(child);
    }
  }
}
```

Naive parallelization

Problem:

Race condition on the global variable `output_list`.

Pseudo code:

```
Node *target;
std::list<Node *> output_list;

void walk(Node *x) {
  if (x->kind == Node::LEAF) {
    if (target->collides_with(x)) {
        output_list.push_back(x);
    }
  } else {
    cilk_for (Node::iterator child = x.begin();
         child != x.end();
         ++child) {
        walk(child);
    }
  }
}
```

Locking solution

Problems:

Lock contention inhibits speedup.

Output order is nondeterministic.

Pseudo code:

```
mutex lock;

void walk(Node *x) {
  if (x->kind == Node::LEAF) {
    if (target->collides_with(x)) {
        lock.acquire();
        output_list.push_back(x);
        lock.release();
    }
  } else {
    cilk_for (Node::iterator child = x.begin();
         child != x.end();
         ++child) {
        walk(child);
    }
  }
}
```

Reducers

Reducer solution:

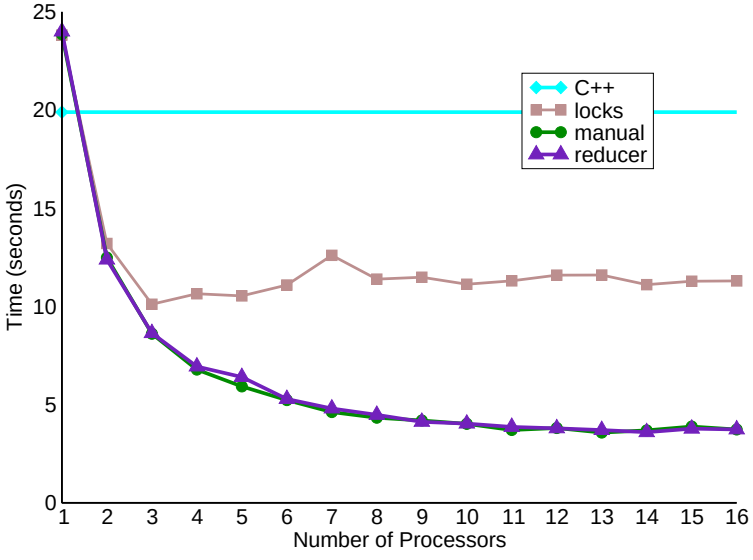
- Define `output_list` as a **reducer**.
- No code restructuring.
- The output is in serial order.
- No locking.
- Low overhead.

Pseudo code:

```
Node *target;
cilk::reducer_list_append<Node *> output_list;

void walk(Node *x) {
  if (x->kind == Node::LEAF) {
    if (target->collides_with(x)) {
        output_list.push_back(x);
    }
  } else {
    cilk_for (
         Node::iterator child = x.begin();
         child != x.end();
         ++child) {
        walk(child);
    }
  }
}
```

Performance of collision detection



Reducers

Properties:

- “Hyperobjects” that support multiple parallel “views”.
- Automatic, user-defined, pairwise merging of views following the fork/join structure of the program.
- Deterministic if the merge operation is associative.
- Multiple reductions can execute in parallel.
- Backward-compatible with sequential semantics (and syntax).

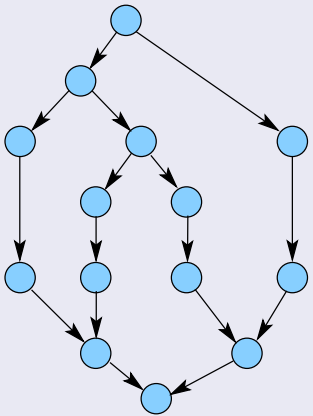
Applications:

- Mapreduce pattern (such as collision detection).
- C++ exceptions.
- Volpack volume rendering.
- File output in bzip2.

The Cilkview parallelism analyzer

What is parallelism?

Dependency graph

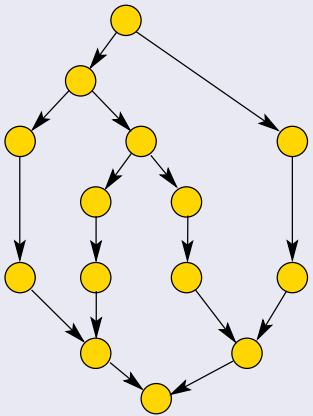


Measures:

- T_P = execution time on P processors

What is parallelism?

Dependency graph

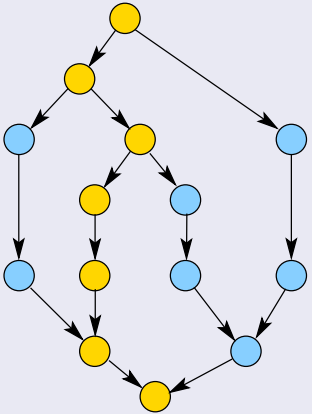


Measures:

- T_P = execution time on P processors
- T_1 = **work**

What is parallelism?

Dependency graph

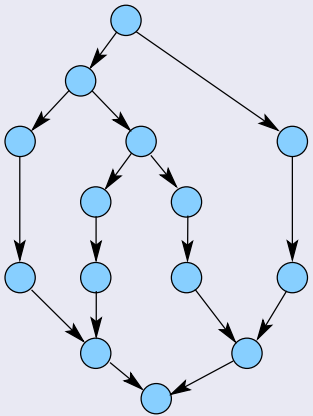


Measures:

- T_P = execution time on P processors
- T_1 = **work**
- T_∞ = **span**

What is parallelism?

Dependency graph



Measures:

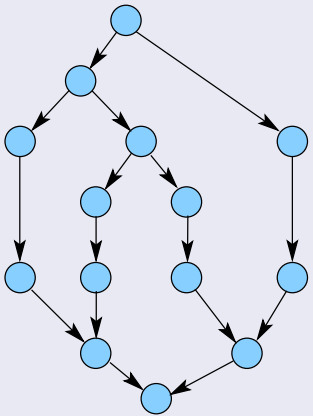
- T_P = execution time on P processors
- T_1 = **work**
- T_∞ = **span**

Work Law:

$$T_P \geq T_1/P.$$

What is parallelism?

Dependency graph



Measures:

- T_P = execution time on P processors
- T_1 = **work**
- T_∞ = **span**

Work Law:

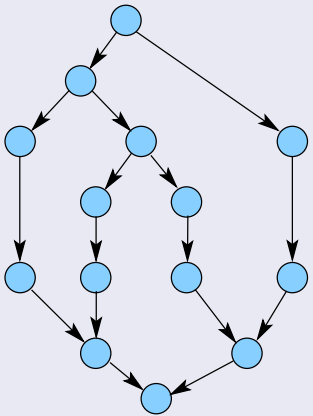
$$T_P \geq T_1/P.$$

Span Law:

$$T_P \geq T_\infty.$$

What is parallelism?

Dependency graph



Measures:

- T_P = execution time on P processors
- T_1 = **work**
- T_∞ = **span**

Work Law:

$$T_P \geq T_1/P.$$

Span Law:

$$T_P \geq T_\infty.$$

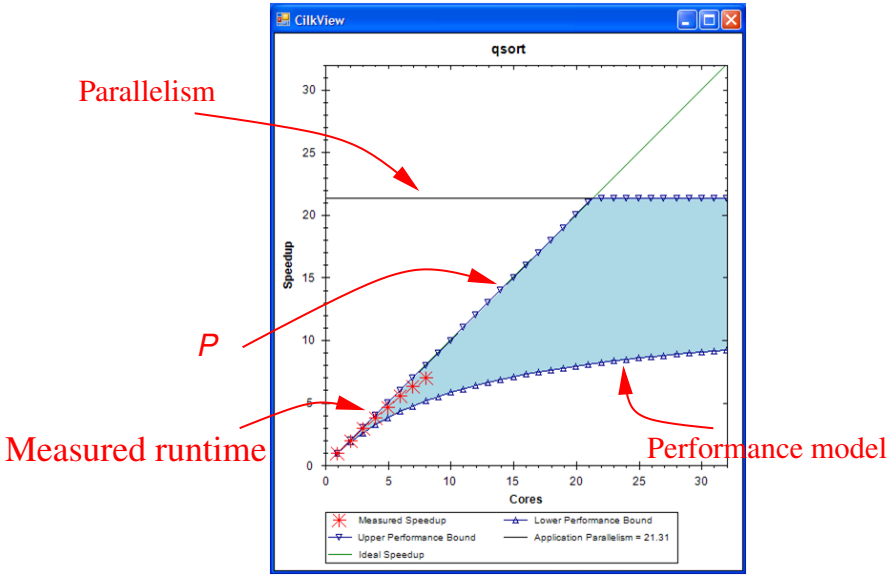
Maximum speedup:

$$\text{speedup} = T_1/T_P \leq T_1/T_\infty = \textbf{parallelism}.$$

The Cilkview parallelism analyzer

- Computes work and span of Cilk programs.
- Instruments production binaries. No debug version required.
- Counts instructions, not time.
- Produces textual and/or graphical output.
- Can measure work and span of portions of a program.
- Fast (about 5x slowdown).
- Negligible memory overhead.

Cilkview screenshot



Where does the performance model come from?

Theorem

Theorem: Cilk's work-stealing scheduler achieves an expected running time of

$$T_P = T_1/P + O(T_\infty)$$

on P processors.

Pseudoproof (not quite correct).

A processor is either working or stealing. The total time all processors spend working is T_1. Each steal has a $1/P$ chance of reducing the span by 1. Thus, the expected number of steals is $O(PT_\infty)$. Since there are P processors, the expected time is

$$(T_1 + O(PT_\infty))/P = T_1/P + O(T_\infty) \ .$$

□

Intel Parallel Building Blocks

- Suite of compilers, libraries and tools for parallelism.
- Cilk keywords for C and C++.
- Cilkscreen, Cilkview.
- Automatic vectorization for SSE/SSE2/AVX/etc.
- Data-parallel array notation: `a[0:n] = b[0:n] + 1`.
- Threading Building Blocks.
- Supports Linux™ and Windows®.

History of Cilk

1992: Theory of the Cilk scheduler. [Blumofe and Leiserson]
1993: Early Cilk implementations on CM-5.
1995: Modern Cilk language. [Blumofe et al.]
1998: Modern Cilk implementation. [Frigo et al.]
1998: Race detector. [Feng and Leiserson]
2004: Adaptive scheduling. [Agrawal]
2005: Exceptions, JCilk. [Danaher et al.]
2007: Cilk Arts founded. [Frigo and Leiserson]
2007: Cilk++ language and implementation.
2009: Cilk Arts acquired by Intel.
2010: Cilk Plus: array notation, integration with Intel tools.

Conclusion

- The Cilk language is a simple expression of fork-join parallelism.
- Cilkscreen detects determinacy races.
- Reducers cure a common set of races.
- Cilkview analyzes the parallelism of your program.
- Cilk Plus is integrated with data-parallel extensions, SIMD instructions, and TBB.
- Emphasis on compatibility with sequential software.

Disclaimer and Attribution

The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions and typographical errors.
The information contained herein is subject to change and may be rendered inaccurate for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product releases, product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. There is no obligation to update or otherwise correct or revise this information. However, we reserve the right to revise this information and to make changes from time to time to the content hereof without obligation to notify any person of such revisions or changes.
NO REPRESENTATIONS OR WARRANTIES ARE MADE WITH RESPECT TO THE CONTENTS HEREOF AND NO RESPONSIBILITY IS ASSUMED FOR ANY INACCURACIES, ERRORS OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION.
ALL IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE ARE EXPRESSLY DISCLAIMED. IN NO EVENT WILL ANY LIABILITY TO ANY PERSON BE INCURRED FOR ANY DIRECT, INDIRECT, SPECIAL OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

The contents of this presentation were provided by individual(s) and/or company listed on the title page. The information and opinions presented in this presentation may not represent AMDs positions, strategies or opinions. Unless explicitly stated, AMD is not responsible for the content herein and no endorsements are implied.