Multicore programming in CilkPlus

Marc Moreno Maza

University of Western Ontario, Canada

CS3350  March 16, 2015
Cilk has been developed since 1994 at the MIT Laboratory for Computer Science by Prof. Charles E. Leiserson and his group, in particular by Matteo Frigo.

Besides being used for research and teaching, Cilk was the system used to code the three world-class chess programs: Tech, Socrates, and Cilkchess.

Over the years, the implementations of Cilk have run on computers ranging from networks of Linux laptops to an 1824-nodes Intel Paragon.

From 2007 to 2009 Cilk has lead to Cilk++, developed by Cilk Arts, an MIT spin-off, which was acquired by Intel in July 2009 and became CilkPlus, see http://www.cilk.com/

CilkPlus can be freely downloaded for Linux as a branch of the gcc compiler collection.

Cilk is still developed at MIT http://supertech.csail.mit.edu/cilk/
CilkPlus (resp. Cilk) is a small set of linguistic extensions to C++ (resp. C) supporting fork-join parallelism.

Both Cilk and CilkPlus feature a provably efficient work-stealing scheduler.

CilkPlus provides a hyperobject library for parallelizing code with global variables and performing reduction for data aggregation.

CilkPlus includes the Cilkscreen race detector and the Cilkview performance analyzer.
Fork-Join Parallelism in CilkPlus

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

- The named child function `cilk_spawn fib(n-1)` may execute in parallel with its parent.
- CilkPlus keywords `cilk_spawn` and `cilk_sync` grant permissions for parallel execution. They do not command parallel execution.
A **scheduler**’s job is to map a computation to particular processors. Such a mapping is called a **schedule**.

- If decisions are made at runtime, the scheduler is *online*, otherwise, it is *offline*.
- CilkPlus’s scheduler maps strands onto processors dynamically at runtime.
The CilkPlus Platform

CilkPlus source

Cilk++ Compiler

Conventional Compiler

Hyperobject Library

Hyperobject Library

Hyperobject Library

Hyperobject Library

Hyperobject Library

Hyperobject Library

Cilkview Scalability Analyzer

Cilkview Scalability Analyzer

Cilkview Scalability Analyzer

Cilkview Scalability Analyzer

Cilkview Scalability Analyzer

Cilkview Scalability Analyzer

Cilkscreen Race Detector

Cilkscreen Race Detector

Cilkscreen Race Detector

Cilkscreen Race Detector

Cilkscreen Race Detector

Cilkscreen Race Detector

Runtime System

Runtime System

Runtime System

Runtime System

Runtime System

Runtime System

Exceptional Performance

Exceptional Performance

Exceptional Performance

Exceptional Performance

Exceptional Performance

Exceptional Performance

Parallel Regression Tests

Parallel Regression Tests

Parallel Regression Tests

Parallel Regression Tests

Parallel Regression Tests

Parallel Regression Tests

Reliable Multi-Threaded Code

Reliable Multi-Threaded Code

Reliable Multi-Threaded Code

Reliable Multi-Threaded Code

Reliable Multi-Threaded Code

Reliable Multi-Threaded Code

Conventional Regression Tests

Conventional Regression Tests

Conventional Regression Tests

Conventional Regression Tests

Conventional Regression Tests

Conventional Regression Tests

Reliable Single-Threaded Code

Reliable Single-Threaded Code

Reliable Single-Threaded Code

Reliable Single-Threaded Code

Reliable Single-Threaded Code

Reliable Single-Threaded Code

Serialization

Serialization

Serialization

Serialization

Serialization

Serialization

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);
    }  
}
The fork-join parallelism model

```
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);
  }
}
```

Example:
fib(4)

"Processor oblivious"

The computation dag unfolds dynamically.

We shall also call this model multithreaded parallelism.
The fork-join parallelism model

$T_p$ is the minimum running time on $p$ processors.

$T_1$ is the sum of the number of instructions at each vertex in the DAG, called the **work**.

$T_\infty$ is the minimum running time with infinitely many processors, called the **span**. This is the length of a path of maximum length from the root to a leaf.

$T_1/T_\infty$: **Parallelism.**

- **Work law**: $T_p \geq T_1/p$.
- **Span law**: $T_p \geq T_\infty$.

Figure: Instruction stream DAG.
In any **greedy schedule**, there are two types of steps:

- **complete step**: There are at least $p$ strands that are ready to run. The greedy scheduler selects any $p$ of them and runs them.
- **incomplete step**: There are strictly less than $p$ threads that are ready to run. The greedy scheduler runs them all.

*For any greedy schedule, we have* \( T_p \leq \frac{T_1}{p} + T_\infty \)
The fork-join multithreaded programming model

**Speedup on** \( p \) **processors**

- \( T_1/T_p \) is called the **speedup on** \( p \) **processors**

- A parallel program execution can have:
  - **linear speedup**: \( T_1/T_P = \Theta(p) \)
  - **superlinear speedup**: \( T_1/T_P = \omega(p) \) (not possible in this model, though it is possible in others)
  - **sublinear speedup**: \( T_1/T_P = o(p) \)
Overheads and burden

Many factors (simplification assumptions of the fork-join parallelism model, architecture limitation, costs of executing the parallel constructs, overheads of scheduling) will make $T_p$ larger in practice than $T_1/p + T_\infty$.

One may want to estimate the impact of those factors:

1. by improving the estimate of the randomized work-stealing complexity result
2. by comparing a CilkPlus program with its C++ elision
3. by estimating the costs of spawning and synchronizing

CilkPlus estimates $T_p$ as $T_p = T_1/p + 1.7 \text{ burden-span}$, where burden-span is 15000 instructions times the number of continuation edges along the critical path.
Cilkview

- Cilkview computes work and span to derive upper bounds on parallel performance.
- Cilkview also estimates scheduling overhead to compute a burdened span for lower bounds.
The cilkview example from the documentation

Using cilk_for to perform operations over an array in parallel:

```c
static const int COUNT = 4;
static const int ITERATION = 1000000;
long arr[COUNT];
long do_work(long k){
    long x = 15;
    static const int nn = 87;
    for (long i = 1; i < nn; ++i)
        x = x / i + k % i;
    return x;
}
int main(){
    for (int j = 0; j < ITERATION; j++)
        cilk_for (int i = 0; i < COUNT; i++)
            arr[i] += do_work( j * i + i + j);
}
1) Parallelism Profile

Work: 6,480,801,250 ins
Span: 2,116,801,250 ins
Burdened span: 31,920,801,250 ins
Parallelism: 3.06
Burdened parallelism: 0.20
Number of spawns/syncs: 3,000,000
Average instructions / strand: 720
Strands along span: 4,000,001
Average instructions / strand on span: 529

2) Speedup Estimate

2 processors: 0.21 - 2.00
4 processors: 0.15 - 3.06
8 processors: 0.13 - 3.06
16 processors: 0.13 - 3.06
32 processors: 0.12 - 3.06
A simple fix

Inverting the two for loops

```c
int main()
{
    cilk_for (int i = 0; i < COUNT; i++)
        for (int j = 0; j < ITERATION; j++)
            arr[i] += do_work( j * i + i + j);
}
```
1) Parallelism Profile

Work : 5,295,801,529 ins
Span : 1,326,801,107 ins
Burdened span : 1,326,830,911 ins
Parallelism : 3.99
Burdened parallelism : 3.99
Number of spawns/syncs: 3
Average instructions / strand : 529,580,152
Strands along span : 5
Average instructions / strand on span: 265,360,221

2) Speedup Estimate

2 processors: 1.40 - 2.00
4 processors: 1.76 - 3.99
8 processors: 2.01 - 3.99
16 processors: 2.17 - 3.99
32 processors: 2.25 - 3.99
## Timing

<table>
<thead>
<tr>
<th>version</th>
<th>#cores = 1</th>
<th>#cores = 2</th>
<th>speedup</th>
<th>#cores = 4</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>original</td>
<td>7.719</td>
<td>9.611</td>
<td>0.803</td>
<td>10.758</td>
<td>0.718</td>
</tr>
<tr>
<td>improved</td>
<td>7.471</td>
<td>3.724</td>
<td>2.006</td>
<td>1.888</td>
<td>3.957</td>
</tr>
</tbody>
</table>