The Serial Equivalence of Intel® Cilk™ Plus

Robert Geva
Parallel Programming Architect
SSG/DPD

Reminder: 3 keywords

```
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;
    }
}
```

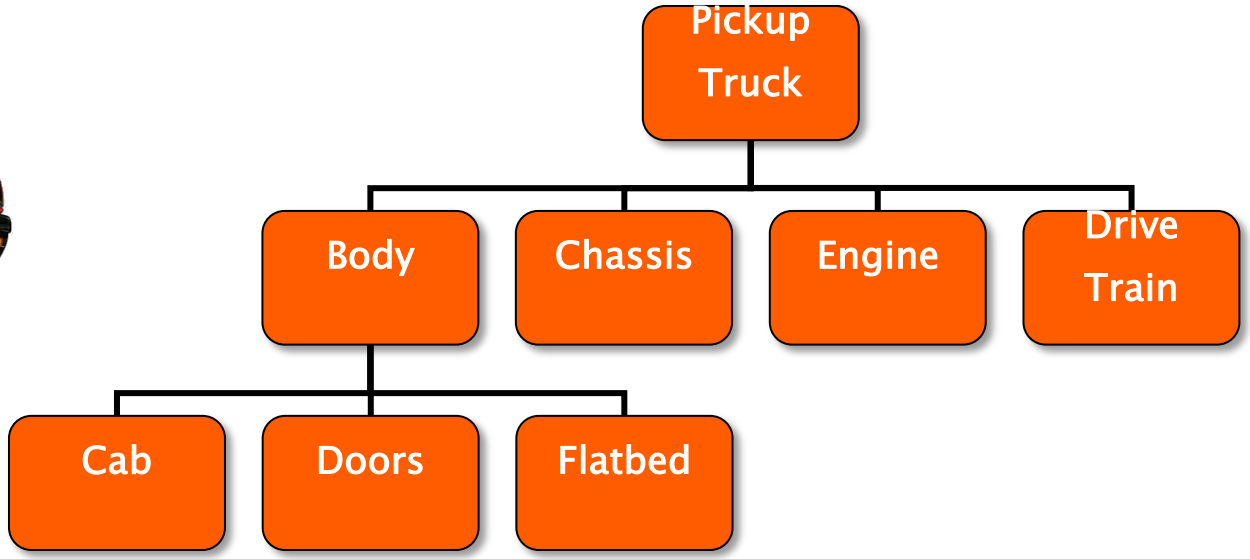
- Spawn a function call
- Sync all child tasks from the same spawning func
- Parallel loop, loop is countable

```
cilk_for (int i = 0; i < max_row; i++) {
    for (int j = 0; j < max_col; j++ ) {
        p[i][j] = mandel( complex(scale(i), scale(j)));
}}
```

Serial Elision

- The serial elision of a Cilk program is well defined:
  - The C/C++ program derived from the Cilk program by
    - cilk_spawn → white space
    - cilk_sync → white space
    - cilk_for → for
- The serial elision of a cilk program is a well formed serial program in C/C++
- A deterministic Cilk program on a single thread behaves the same as its elision
- A cilk program w/o determinacy race behaves the same as its serial elision when running on any number of threads
- For most library solutions (e.g. TBB, PPL), an equivalent property is not well defined
- For OpenMP, the equivalent property does not hold

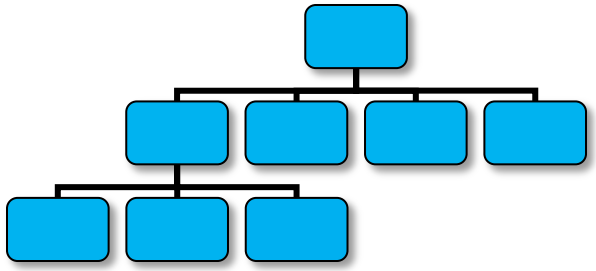
Real-world example: Collision Detection



Goal: Find all "collisions" between an assembly and a target object.

Hyper Objects enable a parallel implementation with serial semantics

Collision Detection, 1



```
std::list<Node *>output_list;
void walk(Node &x, Node &target)) {
    if (x.is_internal())
    {
        cilk_for(Node::iterator child = x.begin();
                                 child != x.end();
                                 ++child) {
            walk(child, target);
        }
    }
    else
        if (target.collides_with(x))
            output_list.push_back(x)
}
```

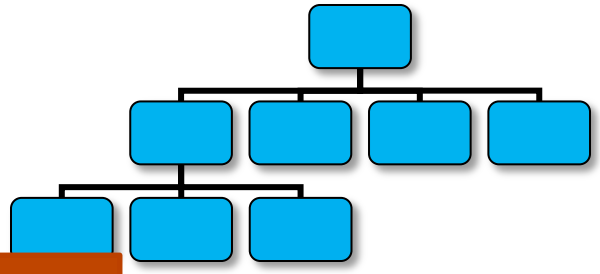
In parallel, traverse tree

At leaf, collect collisions

Parallel update of list is a ***Data Race!***

Data races are almost always bugs

Collision Detection, 2



```
std::list<Node *>output_list;
void walk(Node &x, Node &target)) {
    if (x.is_internal())
    {
        cilk_for(Node::iterator child = x.begin();
                                 child != x.end();
                                 ++child) {
            walk(child, target);
        }
    }
    else
        if (target.collides_with(x))
        {
            m.lock();
            output_list.push_back(x);
            m.unlock();
        }
}
```

In parallel, traverse tree

At leaf, collect collisions

Add lock

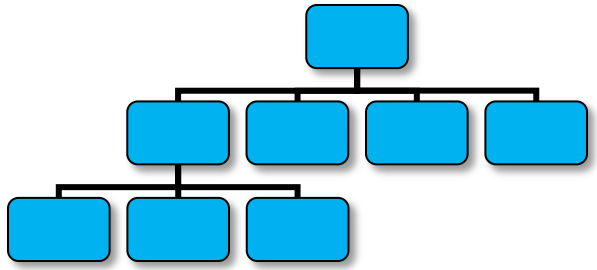
- **Poor performance**
- **Order not deterministic.**

Locks create serial bottlenecks

Collision Detection, 3

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

void walk(Node &x, Node &target)) {
    if (x.is_internal())
    {
        cilk_for(Node::iterator child = x.begin();
                                 child != x.end();
                                 ++child) {
            walk(child, target);
        }
    }
    else
        if (target.collides_with(x))
            output_list.push_back(x);
}
```



In parallel, traverse tree

At leaf, collect collisions

Change list to hyper-object

- Good performance. Serial order!

serial semantics: guaranteed order of nodes in resulting list

Language → serial equivalence

1. Parent Stealing
   1. Spawned child is always scheduled before the continuation
   2. Same order as in the serial execution
2. Arguments evaluated by parent
   1. Side effects are available to the parent
   2. No opportunity to create data races between the evaluaiton of the arguments to the same spawned function
3. Implicit sync →Structured fork – join parallelism
   1. The parents stack is always available while child is executing,
   2. In particular when the parent passes a stack address to the child

```
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 fib (--n);
Cilk_spawn f(g(x),h(y));
```

Legal Disclaimer

- INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL® PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT. INTEL PRODUCTS ARE NOT INTENDED FOR USE IN MEDICAL, LIFE SAVING, OR LIFE SUSTAINING APPLICATIONS.
- Intel may make changes to specifications and product descriptions at any time, without notice.
- All products, dates, and figures specified are preliminary based on current expectations, and are subject to change without notice.
- Intel, processors, chipsets, and desktop boards may contain design defects or errors known as errata, which may cause the product to deviate from published specifications. Current characterized errata are available on request.
- Nehalem, Westmere, Sandy Bridge and other code names featured are used internally within Intel to identify products that are in development and not yet publicly announced for release. Customers, licensees and other third parties are not authorized by Intel to use code names in advertising, promotion or marketing of any product or services and any such use of Intel's internal code names is at the sole risk of the user
- Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance.
- Intel, Core, Itanium and the Intel logo are trademarks of Intel Corporation in the United States and other countries.
- *Other names and brands may be claimed as the property of others.
-

Optimization Notice

Intel® Composer XE 2011 includes compiler options that optimize for instruction sets that are available in both Intel® and non-Intel microprocessors (for example SIMD instruction sets), but do not optimize equally for non-Intel microprocessors. In addition, certain compiler options for Intel® Composer XE 2011 are reserved for Intel microprocessors. For a detailed description of these compiler options, including the instruction sets they implicate, please refer to "Intel® Composer XE 2011 Documentation > Intel® C++ Compiler 12.0 User and Reference Guides > Compiler Options." Many library routines that are part of Intel® Composer XE 2011 are more highly optimized for Intel microprocessors than for other microprocessors. While the compilers and libraries in Intel® Composer XE 2011 offer optimizations for both Intel and Intel-compatible microprocessors, depending on the options you select, your code and other factors, you likely will get extra performance on Intel microprocessors.

While the paragraph above describes the basic optimization approach for Intel® Composer XE 2011, with respect to Intel's compilers and associated libraries as a whole, Intel® Composer XE 2011 may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include Intel® Streaming SIMD Extensions 2 (Intel® SSE2), Intel® Streaming SIMD Extensions 3 (Intel® SSE3), and Supplemental Streaming SIMD Extensions 3 (Intel® SSSE3) instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors.

Intel recommends that you evaluate other compilers to determine which best meet your requirements.