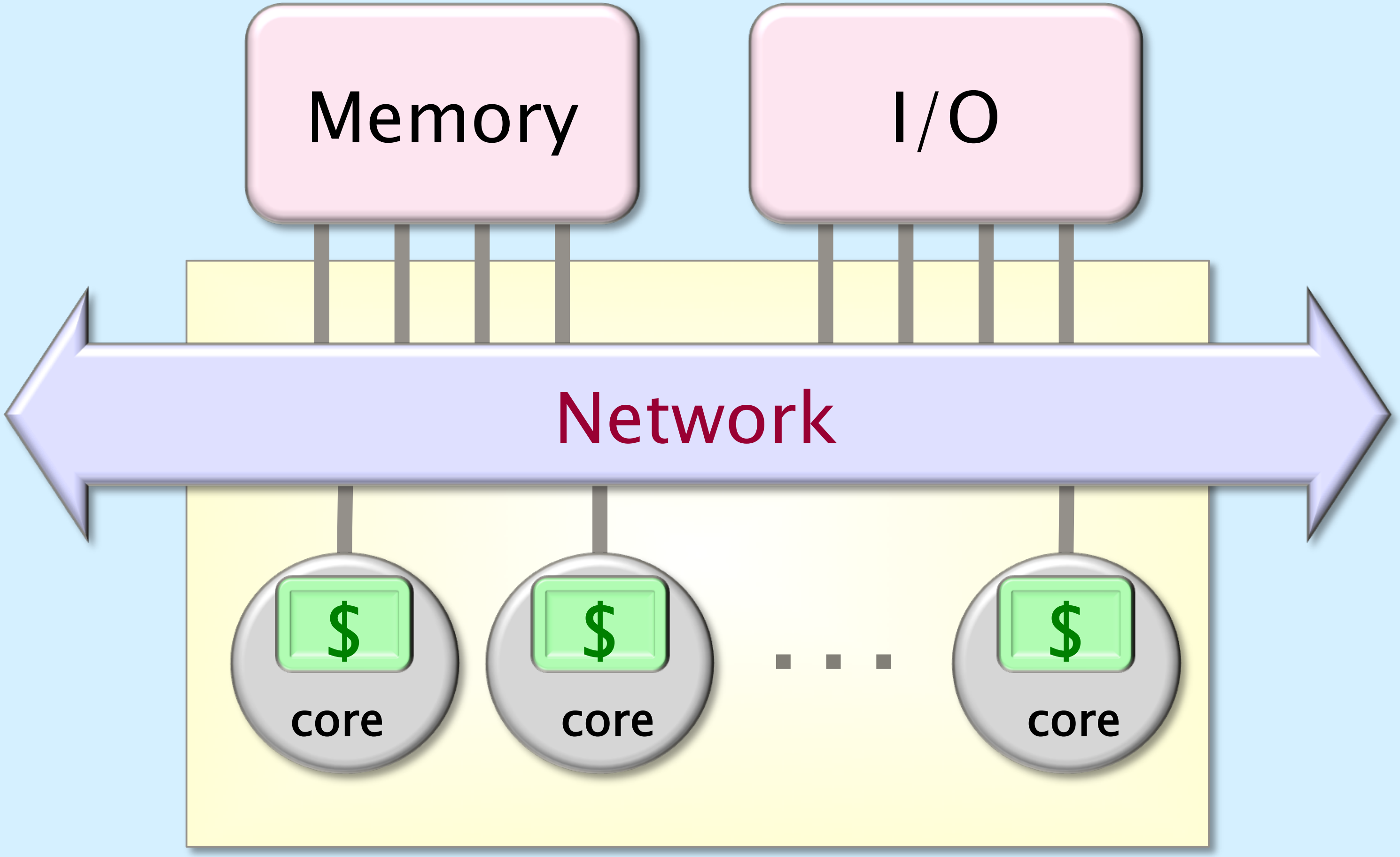
CS 140 : Feb 2, 2015
Multicore (and Shared Memory) Programming with Cilk Plus

- Multicore and shared memory
- Cilk Plus and the divide & conquer paradigm
- Data races
- Analyzing performance in Cilk Plus

Thanks to Charles E. Leiserson for some of these slides

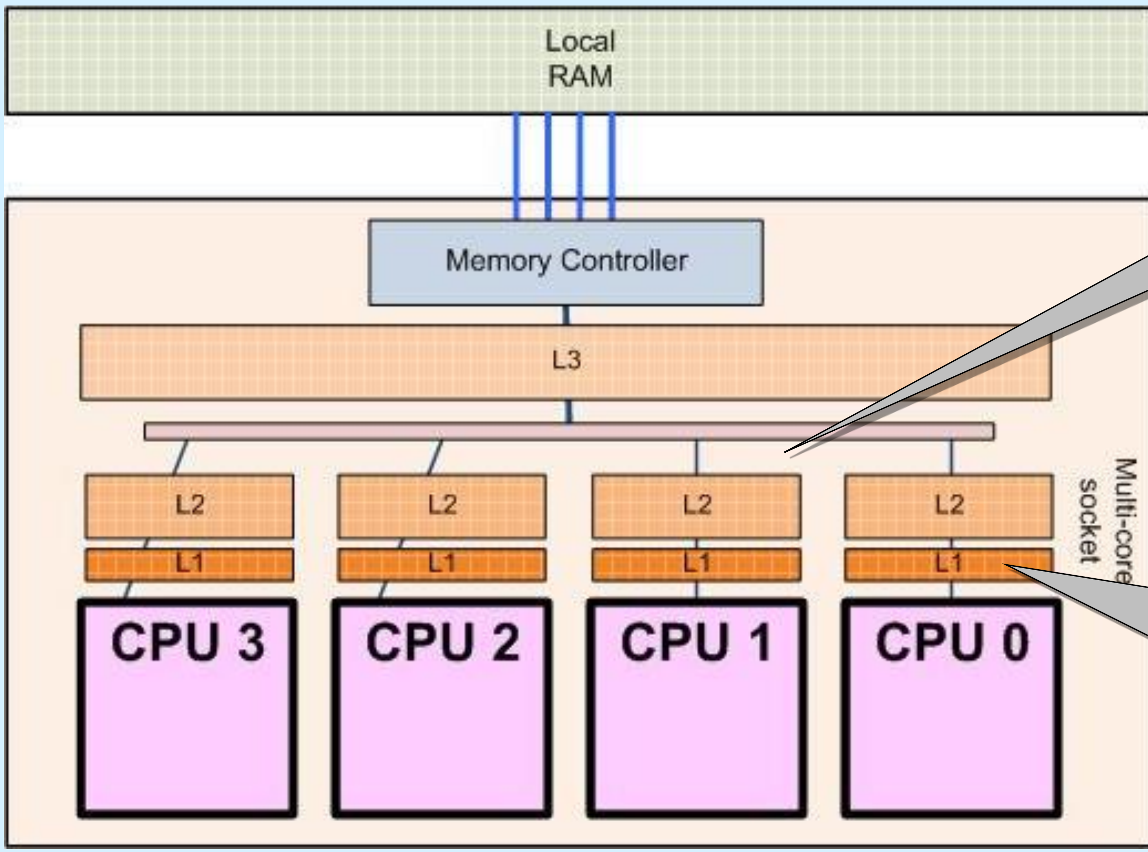
Multicore Architecture



Chip Multiprocessor

Desktop Multicores Today

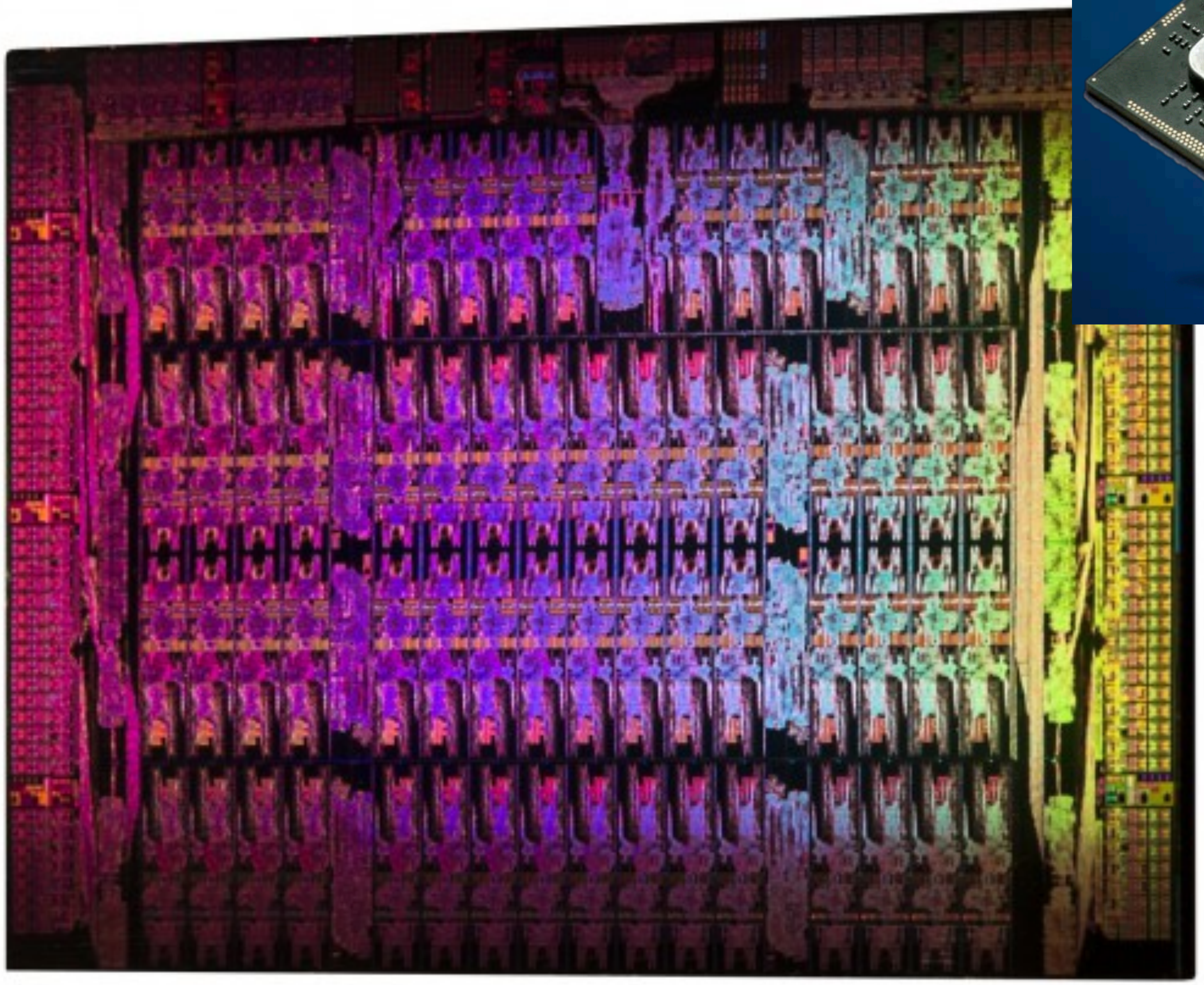
This is your AMD Shangai or Intel Core i7 (Nehalem) !

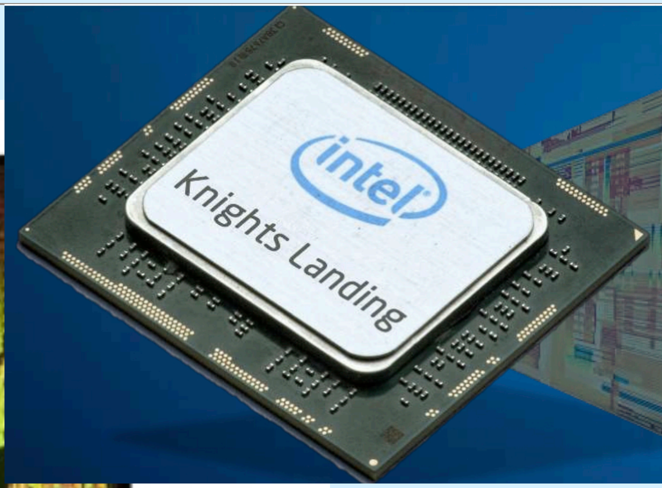


On-chip interconnect

Private cache: Cache coherence is required

62-core Xeon Phi chip





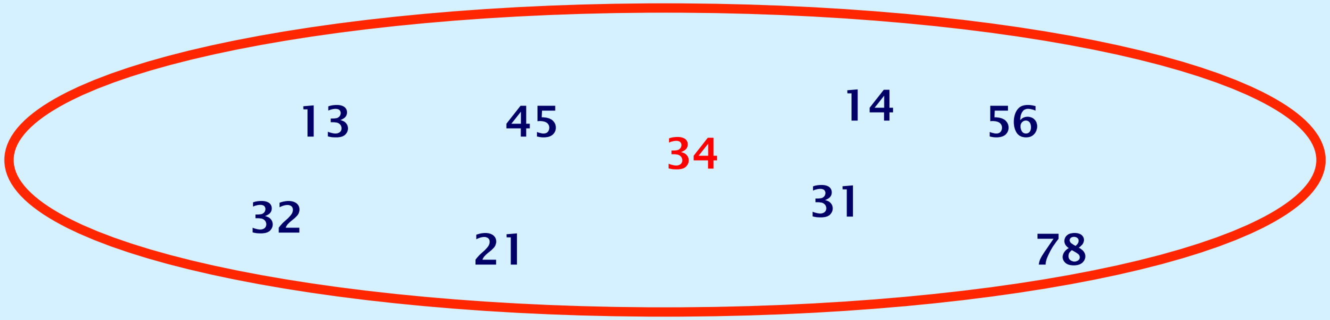
Cilk (Plus)

- Cilk Plus is a faithful extension of C++
- Programs use the **divide-and-conquer** paradigm. Two hints to the compiler:
  - cilk_spawn: *this function can run in parallel with its caller.*
  - cilk_sync: *all spawned children must return before execution passes this point*.
- Third hint for convenience only (compiler converts it to cilk_spawn and cilk_sync)
  - cilk_for: *loop iterations can run in parallel.*
- Cilk also has reducers to avoid data races in global variables.

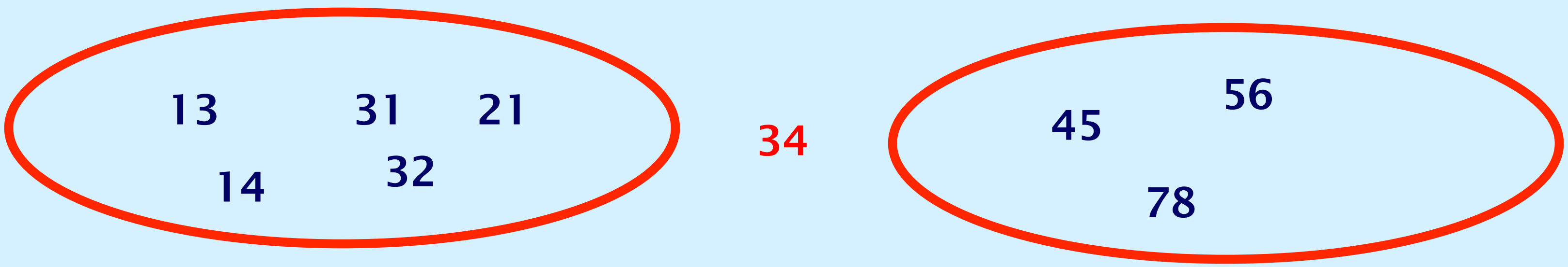
History (and names) of Cilk

- *MIT Cilk: 1994 – 2006*
  - Cilk started as a research project at MIT…
- *Cilk Arts Cilk++: 2006 – 2009*
  - Then Leiserson & co. built a commercial compiler…
- *Intel Cilk++: 2009 – 2010*
  - … then Intel bought Cilk++ from Cilk Arts …
- *Intel Cilk Plus: 2010 – now*
  - … and made it part of "Intel Parallel Building Blocks"
  - Cilk Plus is also a branch of gcc++ now.
- **Intel Cilk Plus is the one you are using on Triton!**
  - There are also free downloads of old Cilk++ around.

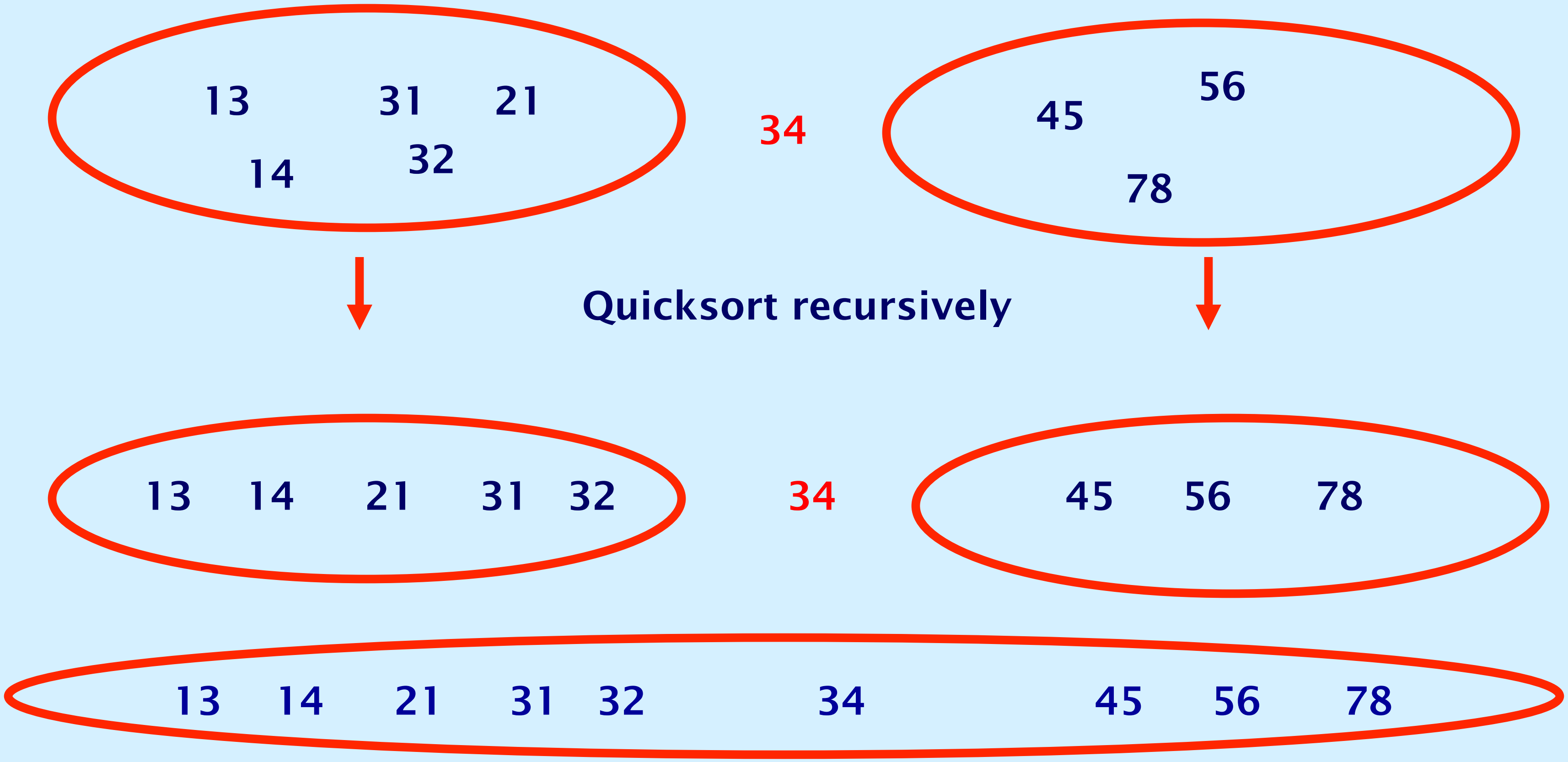
QUICKSORT



Partition around Pivot



QUICKSORT



13 31 21
14 32
34
56
45
78
Quicksort recursively
13 14 21 31 32
34
45 56 78
13 14 21 31 32 34 45 56 78

Nested Parallelism

Example: Quicksort

```
template <typename T>
void qsort(T begin, T end) {

  if (begin != end) {

     T middle = partition(begin, end, …);

     cilk_spawn qsort(begin, middle);

     qsort(max(begin + 1, middle), end);

     cilk_sync;
  }
}
```

Nested Parallelism

Example: Quicksort

```
template <typename T>
void qsort(T begin, T end) {
  if (begin != end) {
    T middle = partition(begin, end, …);
    cilk_spawn qsort(begin, middle);
    qsort(max(begin + 1, middle), end);
    cilk_sync;
  }
}
```

The named *child* function may execute in parallel with the *parent* caller.

Control cannot pass this point until all spawned children have returned.

Cilk Loops

Example: Matrix transpose

```
cilk_for (int i=1; i<n; ++i) {
    cilk_for (int j=0; j<i; ++j) {
        B[i][j] = A[j][i];
  }
}
```

- A `cilk_for` loop's iterations execute in parallel.
- Loop index must be declared in the cilk_for( ).
- End condition is evaluated just once,
  at the beginning of the loop.
- Loop increment must be a **const** value.
- No "break" or "return" allowed inside the loop.

Serial Correctness

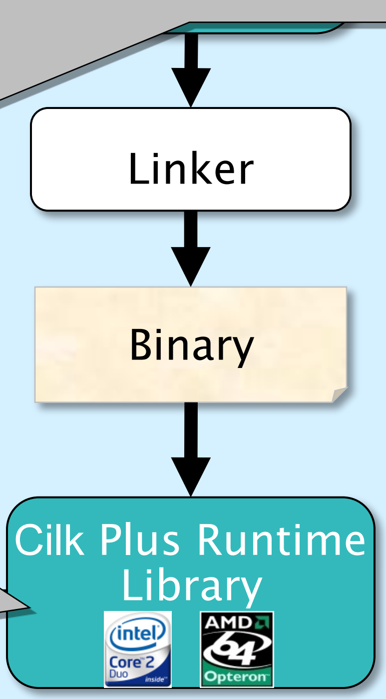
```
int fib (int n) {
  if (n<2) return (n);
  else {
    int x,y;
    x = cilk_spawn fib(n-
    y = fib(n-2);
    cilk_sync;
    return (x+y);
  }
}
```

Cilk source

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

Serialization

The *serialization* is the code with the Cilk keywords replaced by null or **C++** keywords.



Serial correctness can be debugged and verified by running the multithreaded code on a single processor.

Serialization

How to seamlessly switch between serial c++ and parallel cilk plus programs?

```
#ifdef CILKPAR
   #include <cilk.h>
#else
   #define cilk_for for
   #define cilk_main main
   #define cilk_spawn
   #define cilk_sync
#endif
```

Add to the beginning of your program

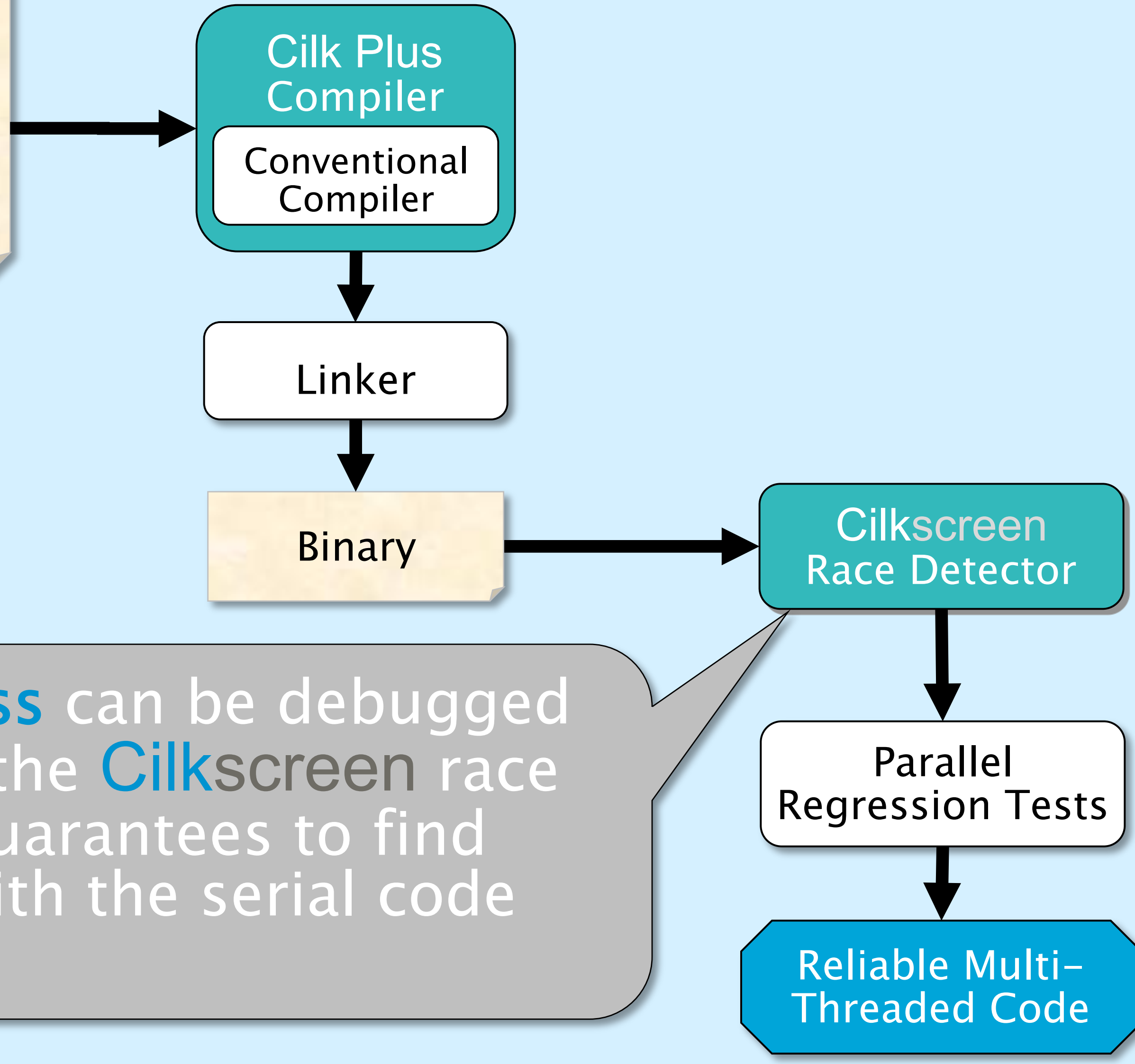
Compile !

- cilk++ -DCILKPAR -O2 -o parallel.exe main.cpp
- g++ -O2 -o serial.exe main.cpp

Parallel Correctness

```
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 source



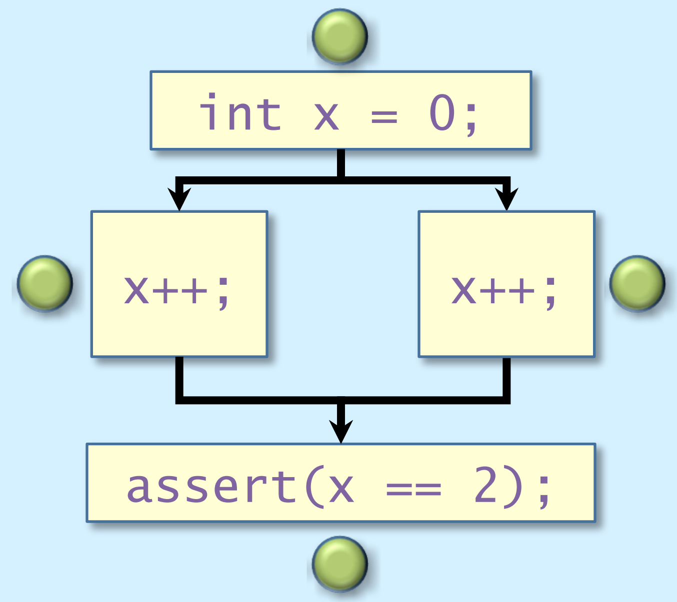
Parallel correctness can be debugged and verified with the Cilkscreen race detector, which guarantees to find inconsistencies with the serial code quickly.

Race Bugs

Definition. A *determinacy race* occurs when two logically parallel instructions access the same memory location and at least one of the instructions performs a write.

Example

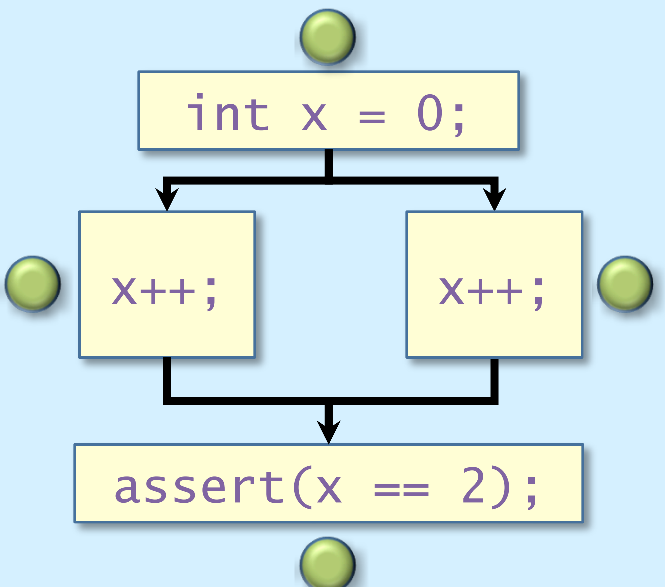
```
int x = 0;
cilk_for(int i=0, i<2, ++i) {
    x++;
}
assert(x == 2);
```

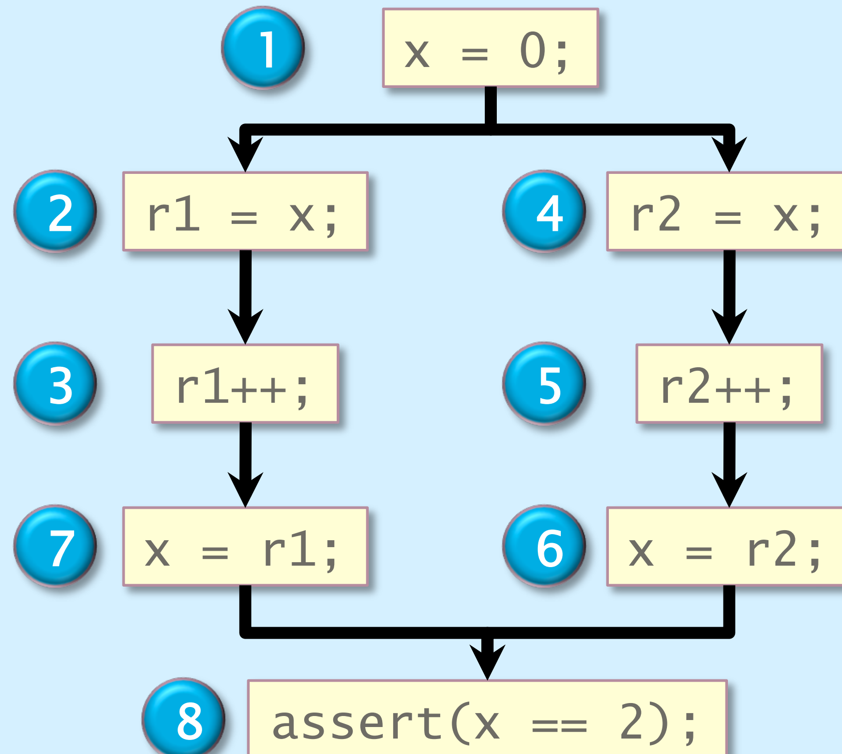


Dependency Graph

Race Bugs

Definition. A *determinacy race* occurs when two logically parallel instructions access the same memory location and at least one of the instructions performs a write.





Types of Races

Suppose that instruction A and instruction B both access a location x, and suppose that A‖B (A is parallel to B).

A	B	Race Type
read	read	none
read	write	read race
write	read	read race
write	write	write race

Two sections of code are *independent* if they have no determinacy races between them.

Avoiding Races

- All the iterations of a `cilk_for` should be independent.
- Between a `cilk_spawn` and the corresponding `cilk_sync`, the code of the spawned child should be independent of the code of the parent, including code executed by additional spawned or called children.

Ex.

```
cilk_spawn qsort(begin, middle);
qsort(max(begin + 1, middle), end);
cilk_sync;
```

Note: The arguments to a spawned function are evaluated in the parent before the spawn occurs.

Cilk Reducers

- A *reducer* is one kind of Cilk *hyperobject*.
- Mostly a solution to global variables, but also broader applications.

```
int result = 0;
cilk_for (int i = 0; i < N; ++i) {
      result += MyFunc(i);
}
```

Data race !

```
#include <cilk/reducer_opadd.h>
…
cilk::reducer< cilk::opadd<int> > result;
cilk_for (int i = 0; i < N; ++i) {
      result += MyFunc(i);
}
```

Race free !

This uses one of the predefined reducers, but you can also write your own reducer easily

Cilk analysis tools

- Cilkscreen race detector:
  - Runs off the executable (compiled specially).
  - Reports *any possibility of a data race* in a particular execution with particular input data.
  - Quite a bit slower than real time.

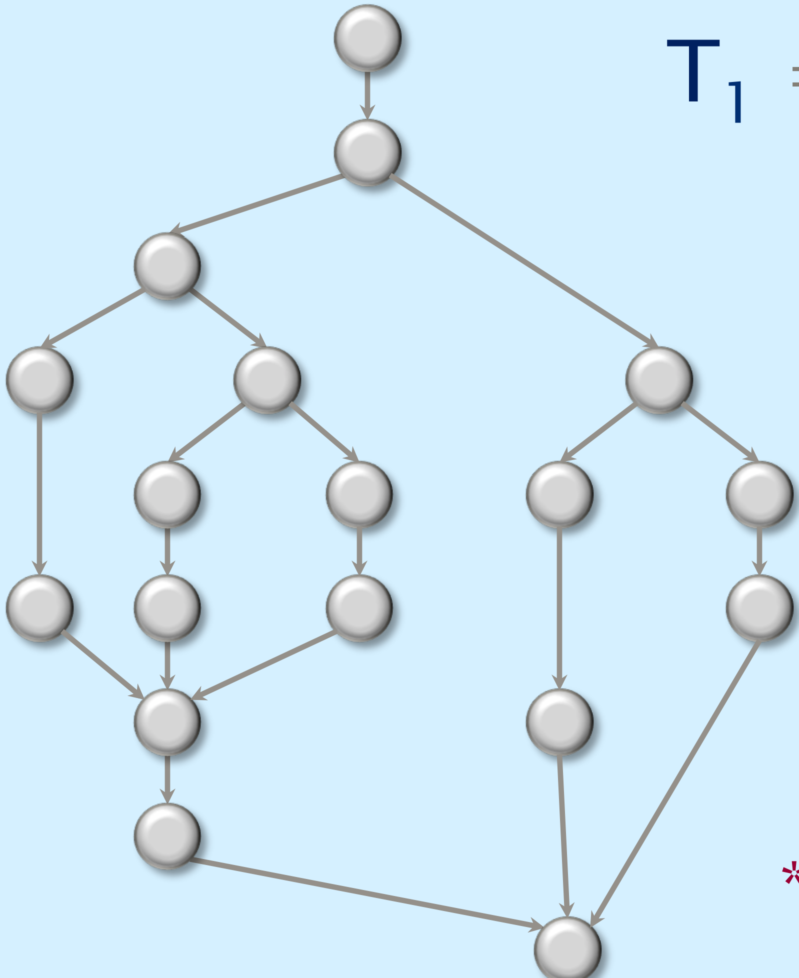
- Cilkview scalability analyzer:
  - Runs off the executable (compiled specially).
  - Reports potential parallelism, burdened parallelism, etc. in theory by counting operations (not by actual clock time); quite a bit slower than real time.
  - Compare results to measured clock times to understand the scaling of your code.

Cilkscreen

- Cilkscreen runs off the binary executable:
  - Compile your program with the `-fcilkscreen` option to include debugging information.
  - Go to the directory with your executable and execute `cilkscreen` *`your_program`* `[`*`options`*`]`
  - Cilkscreen prints information about any races it detects.
- For a given input, Cilkscreen mathematically guarantees to localize a race if there exists a parallel execution that could produce results different from the serial execution.
- It runs about 20 times slower than real-time.

Complexity Measures

T_P = execution time on P processors



T_1 = ***work*** T_∞ = ***span****

WORK LAW

- $T_P \geq T_1/P$

SPAN LAW

- $T_P \geq T_\infty$

*Also called ***critical-path length*** or ***computational depth***.

Speedup

Def. T_1/T_P = ***speedup*** on P processors.

If $T_1/T_P = \Theta(P)$, we have ***linear speedup***,

$= P$, we have ***perfect linear speedup***,

$> P$, we have ***superlinear speedup***,

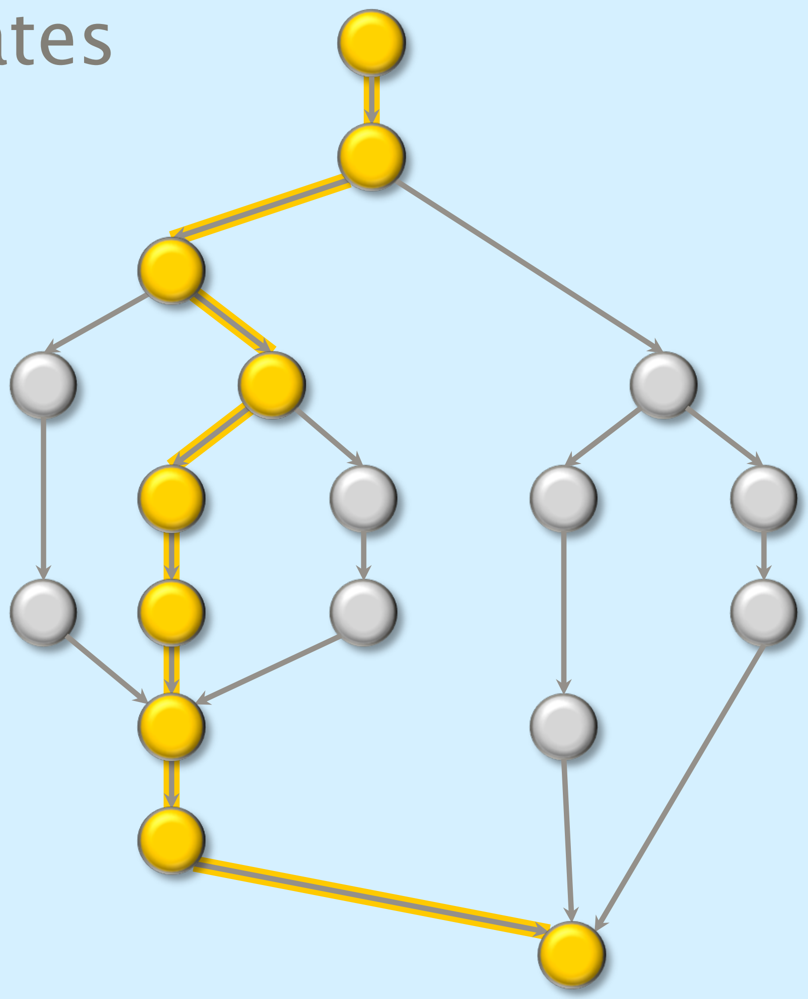
which is not possible in this performance model, because of the Work Law $T_P \geq T_1/P$.

(Potential) Parallelism

Because the Span Law dictates that $T_P \geq T_\infty$, the maximum possible speedup given T_1 and T_∞ is

T_1/T_∞ = *parallelism*

= the average amount of work per step along the span.



Three Tips on Parallelism

1. *Minimize the span* to maximize parallelism. Try to generate 10 times more parallelism than processors for near-perfect linear speedup.
2. If you have plenty of parallelism, try to trade some of it off for *reduced work overheads*.
3. Use *divide-and-conquer recursion* or *parallel loops* rather than spawning one small thing off after another.

Do this:

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

Not this:

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

Three Tips on Overheads

1. Make sure that work/#spawns is not too small.
   - Coarsen by using function calls and *inlining* near the leaves of recursion rather than spawning.
2. Parallelize *outer loops* if you can, not inner loops. If you must parallelize an inner loop, coarsen it, but not too much.
   - 500 iterations should be plenty coarse for even the most meager loop.
   - Fewer iterations should suffice for "fatter" loops.
3. Use *reducers* only in sufficiently fat loops.