NUWEST
Jan. 18, 2024

DEMO

Writing Fast Task-Parallel Code Using OpenCilk

Tao B. Schardl

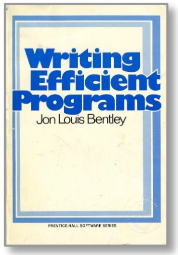
Based on slides and materials from MIT 6.106 lecturers.

Teaching Software Performance Engineering

MIT 6.106: Software Performance Engineering

- Upper-level undergraduate 1-semester class
- ~140 students per year
- Taught using C and OpenCilk
- Prerequisites: algorithms, programming, computer architecture

Lecture topics include:

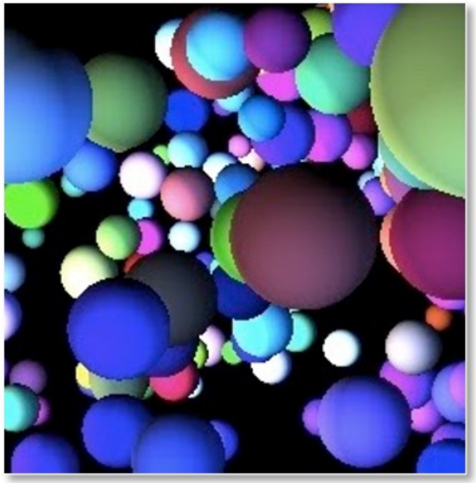


- Bentley rules
- Bit hacks
- Assembly language and computer architecture
- Cache-efficient algorithms
- Measurement and timing
- Task parallelism
- Nondeterministic parallel programming
- And more!

6.106 Projects

In 6.106, students primarily work on 4 open-ended projects.

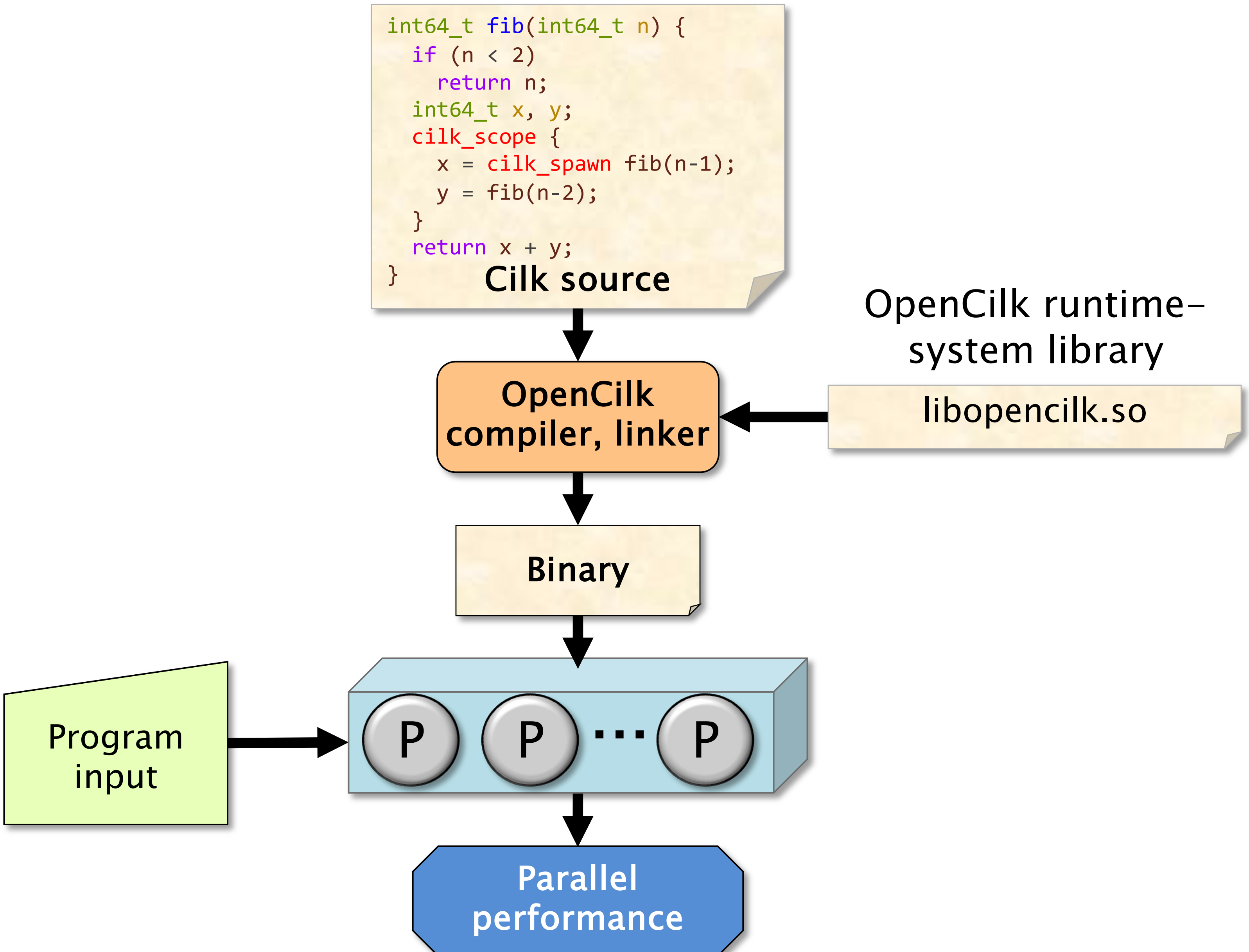
- Students are given a correct, but slow, C program to solve a problem.
- Students are charged with making that program run as fast as possible on a shared-memory multicore.
- Some projects involve only serial performance optimizations.
- Others involve parallel programming using OpenCilk.



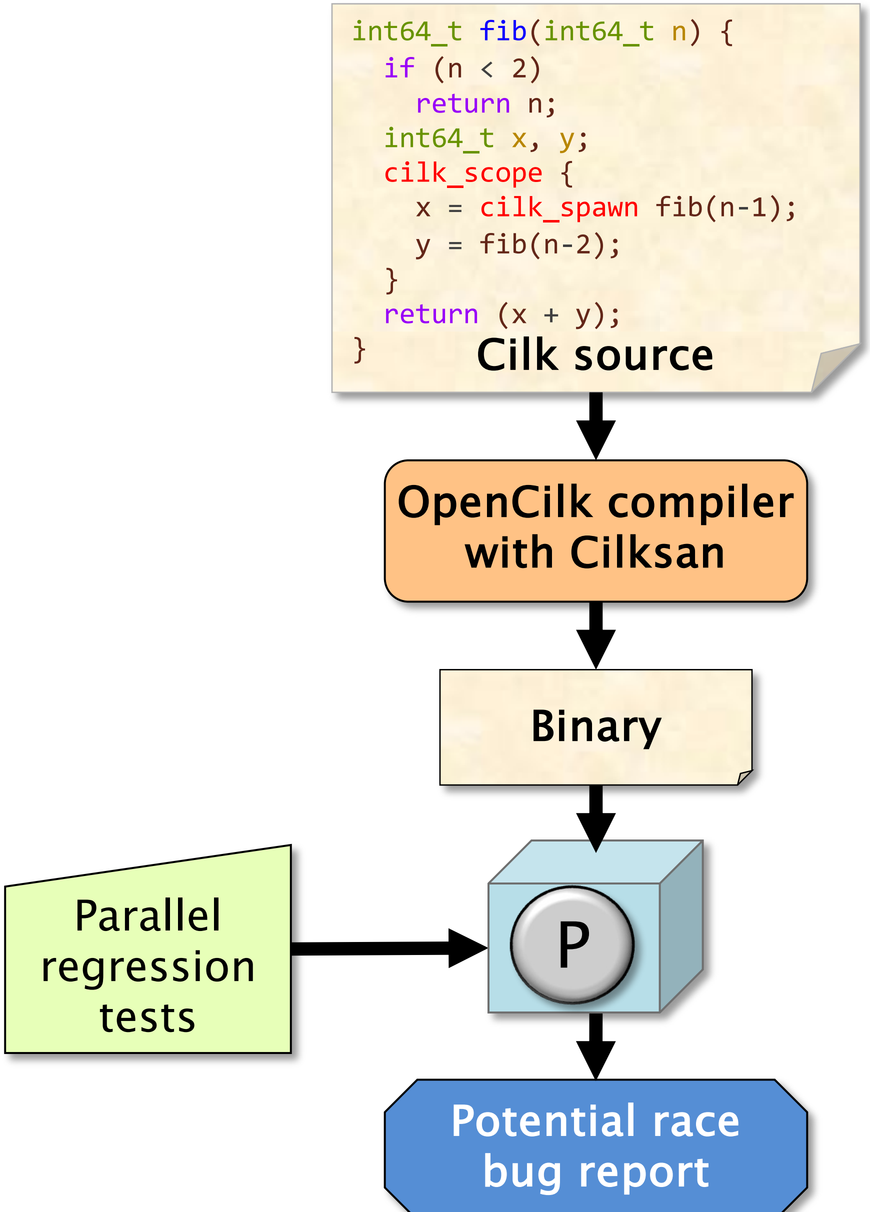
Example project:

Simulation and rendering of colliding spheres

OpenCilk Platform



Parallel Testing



Cilksan finds and localizes race bugs.

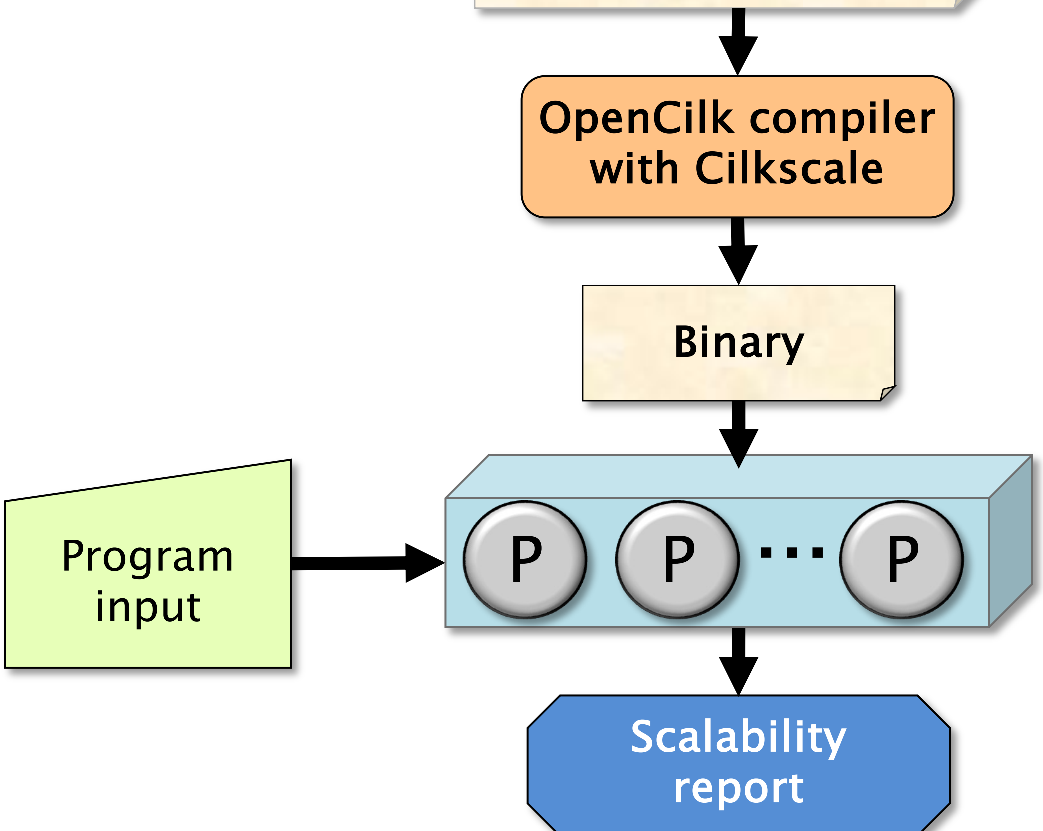
- If an ostensibly deterministic Cilk program could possibly behave nondeterministically on a given input, Cilksan guarantees to report and localize the offending race.
- Cilksan employs a regression-test methodology, where the programmer provides test inputs.

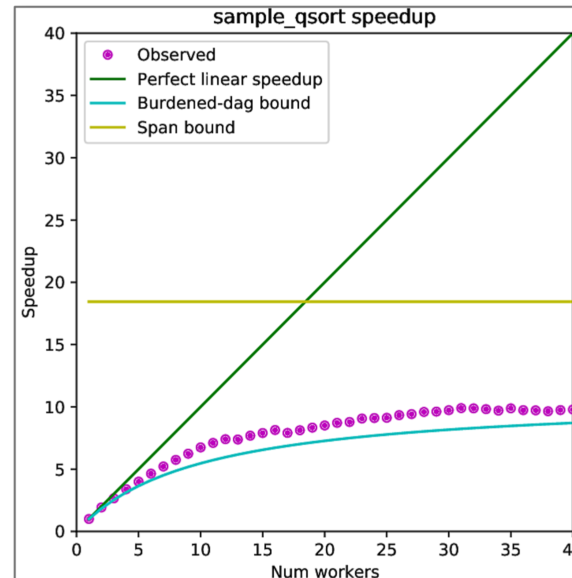
Scalability Analysis

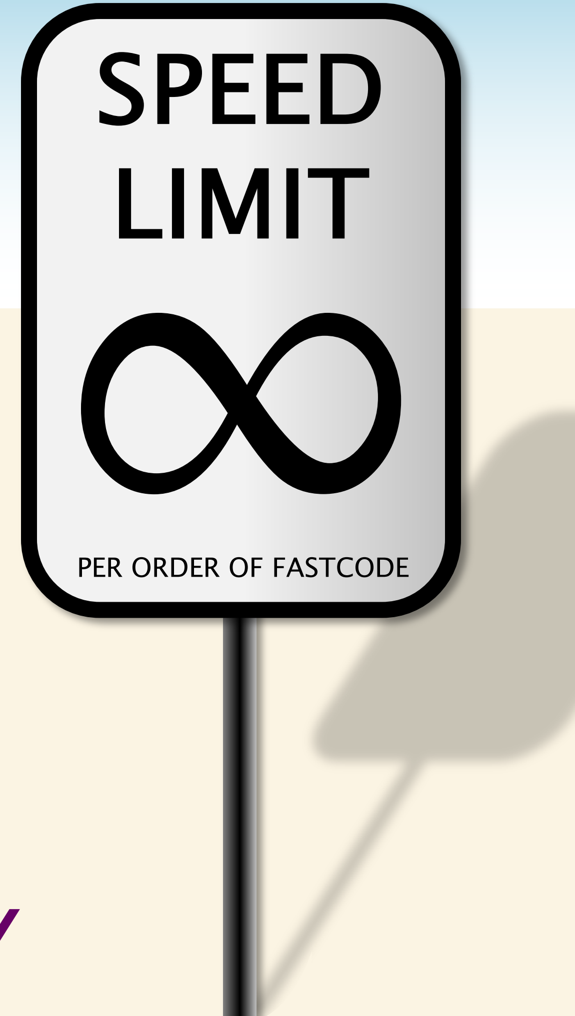
```
int64_t fib(int64_t n) {
  if (n < 2)
    return n;
  int64_t x, y;
  cilk_scope {
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
  }
  return (x + y);
}
```

Cilk source

Cilkscale analyzes how well your program will scale to larger machines.







Lecture 1 Case Study
Matrix Multiplication

Square-Matrix Multiplication

$$\begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{pmatrix}$$

C A B

$$\boxed{c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}}$$

Assume for simplicity that $n = 2^k$.

AWS c4.8xlarge Machine Specs

Feature	Specification
Microarchitecture	Haswell (Intel Xeon E5-2666 v3)
Clock frequency	2.9 GHz
Processor chips	2
Processing cores	9 per processor chip
Hyperthreading	2 way
Floating-point unit	8 double-precision operations, including fused-multiply-add, per core per cycle
Cache-line size	64 B
L1-icache	32 KB private 8-way set associative
L1-dcache	32 KB private 8-way set associative
L2-cache	256 KB private 8-way set associative
L3-cache (LLC)	25 MB shared 20-way set associative
DRAM	60 GB

Peak $= (2.9 \times 10^9) \times 2 \times 9 \times 16 = 836$ GFLOPS

Version 1: Nested Loops in Python

```
import sys, random
from time import *

n = 4096

A = [[random.random()
      for row in xrange(n)]
     for col in xrange(n)]
B = [[random.random()
      for row in xrange(n)]
     for col in xrange(n)]
C = [[0 for row in xrange(n)]
     for col in xrange(n)]

start = time()
for i in xrange(n):
    for j in xrange(n):
        for k in xrange(n):
            C[i][j] += A[i][k] * B[k][j]
end = time()

print '%0.6f' % (end - start)
```

Running time:

≈ 6 microseconds?

≈ 6 milliseconds?

≈ 6 seconds?

≈ 6 hours?

≈ 6 days?

Version 1: Nested Loops in Python

```
import sys, random
from time import *

n = 4096

A = [[random.random()
      for row in xrange(n)]
     for col in xrange(n)]
B = [[random.random()
      for row in xrange(n)]
     for col in xrange(n)]
C = [[0 for row in xrange(n)]
     for col in xrange(n)]

start = time()
for i in xrange(n):
    for j in xrange(n):
        for k in xrange(n):
            C[i][j] += A[i][k] * B[k][j]
end = time()

print '%0.6f' % (end - start)
```

Running time:
= 21042 seconds
≈ 6 hours

Is this fast?

How fast can we make this code through software performance engineering?

After Optimizations

Version	Implementation	Running time (s)	Relative speedup	Absolute Speedup	GFLOPS	Percent of peak
1	Python	21041.67	1.00	1	0.006	0.001
2	Java	2387.32	8.81	9	0.058	0.007
3	C	1155.77	2.07	18	0.118	0.014
4	+ interchange loops	177.68	6.50	118	0.774	0.093
5	+ optimization flags	54.63	3.25	385	2.516	0.301
6	Parallel loops	3.04	17.97	6,921	45.211	5.408
7	Parallel divide-and-conquer	1.30	1.38	16,197	105.722	12.646
8	+ compiler vectorization	0.70	1.87	30,272	196.341	23.486
9	+ AVX intrinsics	0.39	1.76	53,292	352.408	41.677
10	Intel MKL	0.41	0.97	51,497	335.217	40.098

Our Version 9 is competitive with Intel's professionally engineered Math Kernel Library!

Optimizing Matrix Multiplication Using OpenCilk

Follow Along Using SpeedCode

SpeedCode provides an online platform to practice programming that focuses on software performance engineering.

- SpeedCode problems are small programming exercises that require performance engineering to solve.
- SpeedCode provides users with an environment that enables software performance engineering, including
  - Access to performance-engineering tools, and
  - Support for parallel programming using OpenCilk.

Available from http://speedcode.org/
Today, we'll use the "Matrix multiplication" problem.

SpeedCode's development is being led by Dr. Tim Kaler.

Our Starting Point

```c
#include <stdlib.h>
#include <stdio.h>
#include <sys/time.h>

#define n 4096
double A[n][n];
double B[n][n];
double C[n][n];

float tdiff(struct timeval *start,
            struct timeval *end) {
  return (end->tv_sec-start->tv_sec) +
    1e-6*(end->tv_usec-start->tv_usec);
}

int main(int argc, const char *argv[]) {
  for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
      A[i][j] = (double)rand() / (double)RAND_MAX;
      B[i][j] = (double)rand() / (double)RAND_MAX;
      C[i][j] = 0;
    }
  }

  struct timeval start, end;
  gettimeofday(&start, NULL);

  for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
      for (int k = 0; k < n; ++k) {
        C[i][j] += A[i][k] * B[k][j];
      }
    }
  }

  gettimeofday(&end, NULL);
  printf("%0.6f\n", tdiff(&start, &end));
  return 0;
}
```

Using the Clang/LLVM 5.0 compiler

Running time = 1,156 seconds
≈ 19 minutes,
or about 2× faster than Java and about 18× faster than Python.

```c
for (int i = 0; i < n; ++i) {
  for (int j = 0; j < n; ++j) {
    for (int k = 0; k < n; ++k) {
      C[i][j] += A[i][k] * B[k][j];
    }
  }
}
```

Loop Order

We can change the order of the loops in this program without affecting its correctness.

```
for (int i = 0; i < n; ++i) {
  for (int j = 0; j < n; ++j) {
    for (int k = 0; k < n; ++k) {
      C[i][j] += A[i][k] * B[k][j];
    }
  }
}
```

Loop Order

We can change the order of the loops in this program without affecting its correctness.

```
for (int i = 0; i < n; ++i) {
    for (int k = 0; k < n; ++k) {
  for (int j = 0; j < n; ++j) {
      C[i][j] += A[i][k] * B[k][j];
    }
  }
}
```

Does the order of loops matter for performance?

Performance of Different Loop Orders

Loop order (outer to inner)	Running time (s)
i, j, k	1155.77
i, k, j	177.68
j, i, k	1080.61
j, k, i	3056.63
k, i, j	179.21
k, j, i	3032.82

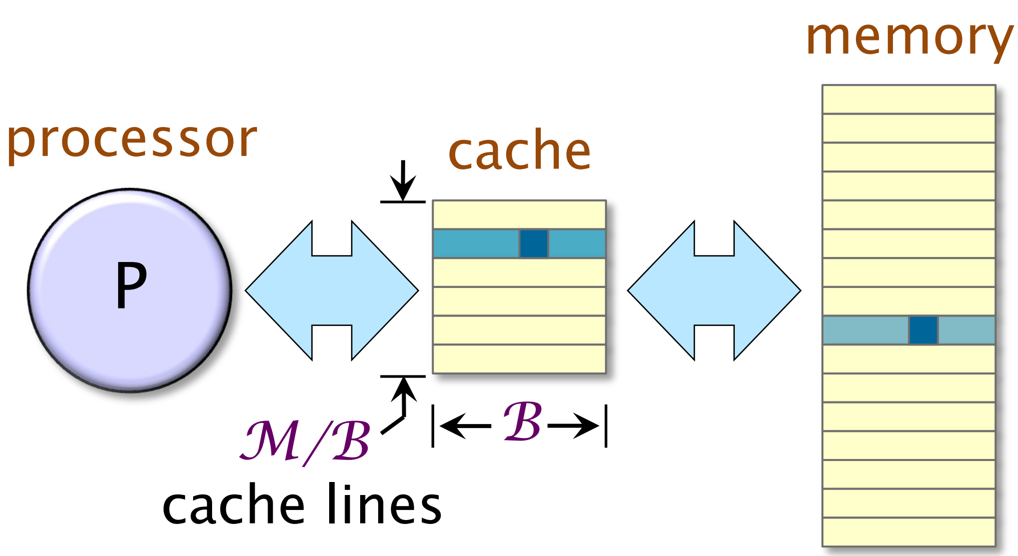
Loop order affects running time by a factor of 18!

What's going on?

Hardware Caches

Each processor reads and writes main memory in contiguous blocks, called *cache lines*.

- Previously accessed cache lines are stored in a smaller memory, called a *cache*, that sits near the processor.
- *Cache hits* — accesses to data in cache — are fast.
- *Cache misses* — accesses to data not in cache — are slow.



Performance of Different Orders

We can measure the effect of different access patterns using the Cachegrind cache simulator:

```
$ valgrind --tool=cachegrind ./mm
```

Loop order (outer to inner)	Running time (s)	Last-level-cache miss rate
i, j, k	1155.77	7.7%
i, k, j	177.68	1.0%
j, i, k	1080.61	8.6%
j, k, i	3056.63	15.4%
k, i, j	179.21	1.0%
k, j, i	3032.82	15.4%

Version 4: Interchange Loops

Version	Implementation	Running time (s)	Relative speedup	Absolute Speedup	GFLOPS	Percent of peak
1	Python	21041.67	1.00	1	0.006	0.001
2	Java	2387.32	8.81	9	0.058	0.007
3	C	1155.77	2.07	18	0.118	0.014
4	+ interchange loops	177.68	6.50	118	0.774	0.093

Compiler Optimization

Clang provides a collection of optimization switches. You can specify a switch to the compiler to ask it to optimize.

Opt. level	Meaning	Time (s)
`-O0`	Do not optimize	177.54
`-O1`	Optimize	66.24
`-O2`	Optimize even more	54.63
`-O3`	Optimize yet more	55.58

Clang also supports optimization levels for special purposes, such as `-Os`, which aims to limit code size, and `-Og`, for debugging purposes.

Version 5: Optimization Flags

Version	Implementation	Running time (s)	Relative speedup	Absolute Speedup	GFLOPS	Percent of peak
1	Python	21041.67	1.00	1	0.006	0.001
2	Java	2387.32	8.81	9	0.058	0.007
3	C	1155.77	2.07	18	0.118	0.014
4	+ interchange loops	177.68	6.50	118	0.774	0.093
5	+ optimization flags	54.63	3.25	385	2.516	0.301

With simple code and compiler technology, we can achieve 0.3% of the peak performance of the machine.

Let's try this on SpeedCode!

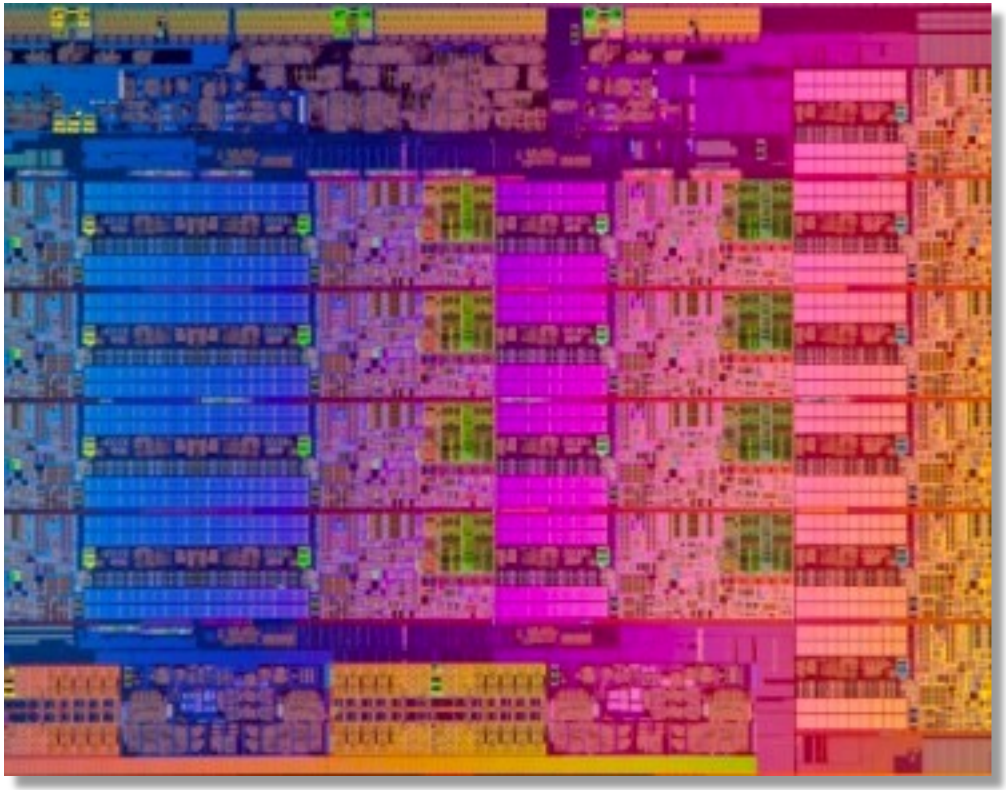
Version 5: Optimization Flags

Version	Implementation	Running time (s)	Relative speedup	Absolute Speedup	GFLOPS	Percent of peak
1	Python	21041.67	1.00	1	0.006	0.001
2	Java	2387.32	8.81	9	0.058	0.007
3	C	1155.77	2.07	18	0.118	0.014
4	+ interchange loops	177.68	6.50	118	0.774	0.093
5	+ optimization flags	54.63	3.25	385	2.516	0.301

With simple code and compiler technology, we can achieve 0.3% of the peak performance of the machine.

Where can we get more performance?

Multicore Parallelism



Intel Haswell E5:
9 cores per chip

The AWS test machine has 2 of these chips.

We're running on just 1 of the 18 parallel-processing cores on this system. *Let's use them all!*

Parallel Loops

Let’s use OpenCilk to parallelize this simple code.

```
cilk_for (int i = 0; i < n; ++i)
  for (int k = 0; k < n; ++k)
    for (int j = 0; j < n; ++j)
      C[i][j] += A[i][k] * B[k][j];
```

Allows all loop iterations to execute in parallel.

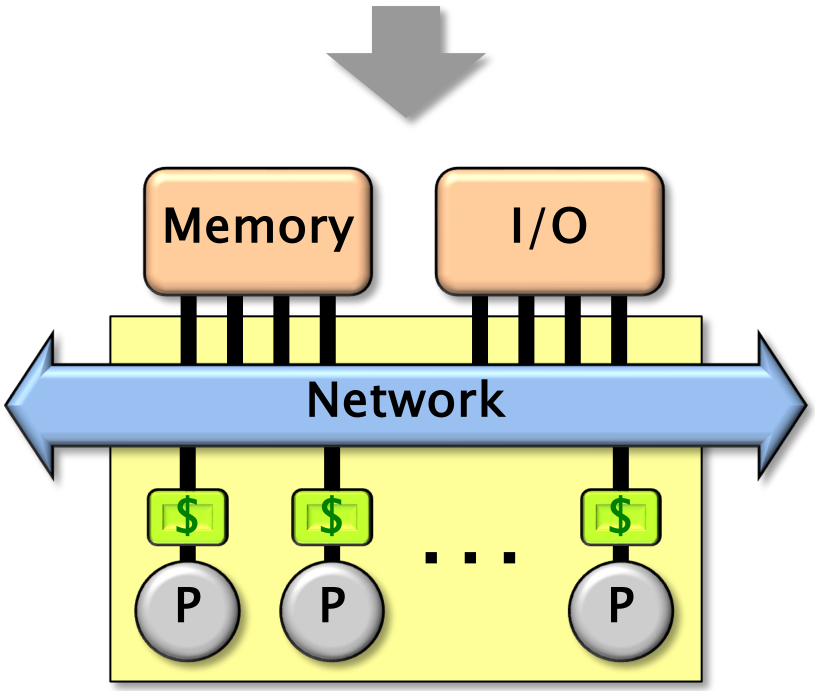
Version	Implementation	Running time (s)	Relative speedup	Absolute Speedup	GFLOPS	Percent of peak
1	Python	21041.67	1.00	1	0.006	0.001
2	Java	2387.32	8.81	9	0.058	0.007
3	C	1155.77	2.07	18	0.118	0.014
4	+ interchange loops	177.68	6.50	118	0.774	0.093
5	+ optimization flags	54.63	3.25	385	2.516	0.301
6	Parallel loops	3.04	17.97	6,921	45.211	5.408

Almost 18x speedup on 18 cores!

OpenCilk Scheduling

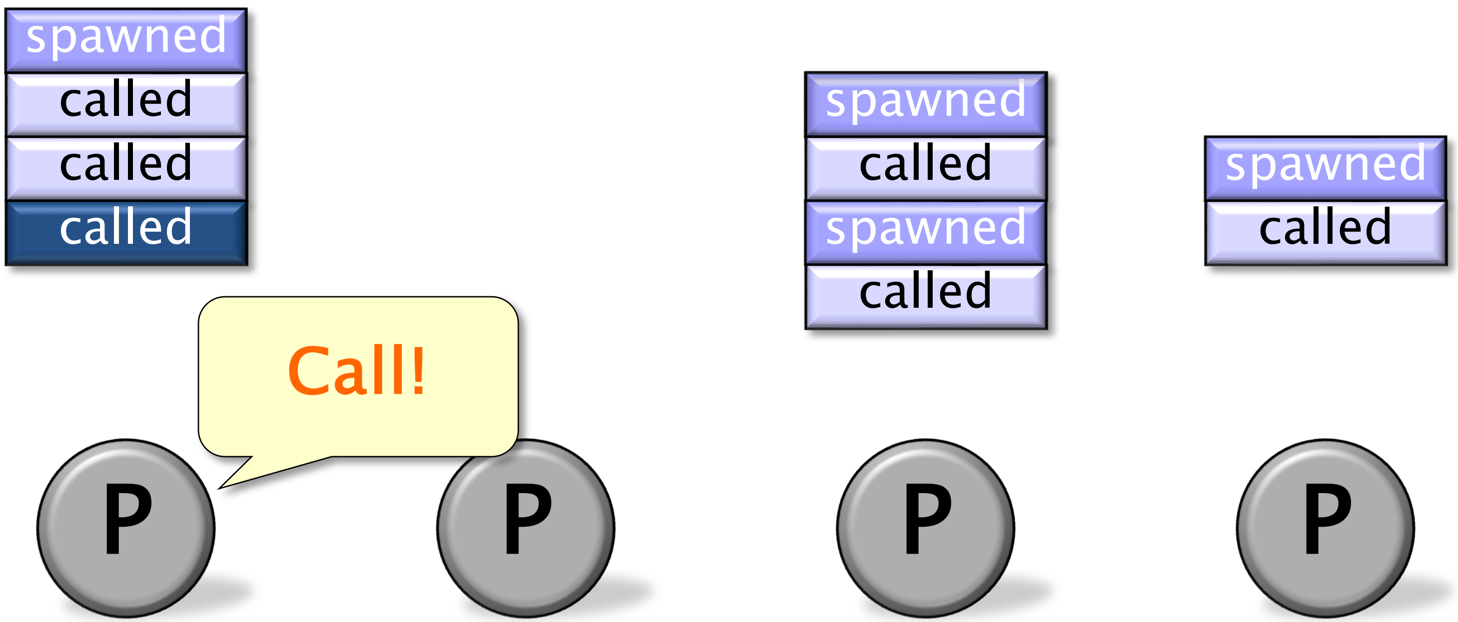
- Cilk allows the programmer to express logical parallelism in an application, in a processor-oblivious fashion.
- The Cilk scheduler maps the executing program onto the processor cores dynamically at runtime.
- Cilk's work-stealing scheduling algorithm is provably efficient.

```
cilk_for (int i = 0; i < n; ++i)
  for (int k = 0; k < n; ++k)
    for (int j = 0; j < n; ++j)
      C[i][j] += A[i][k] * B[k][j];
```



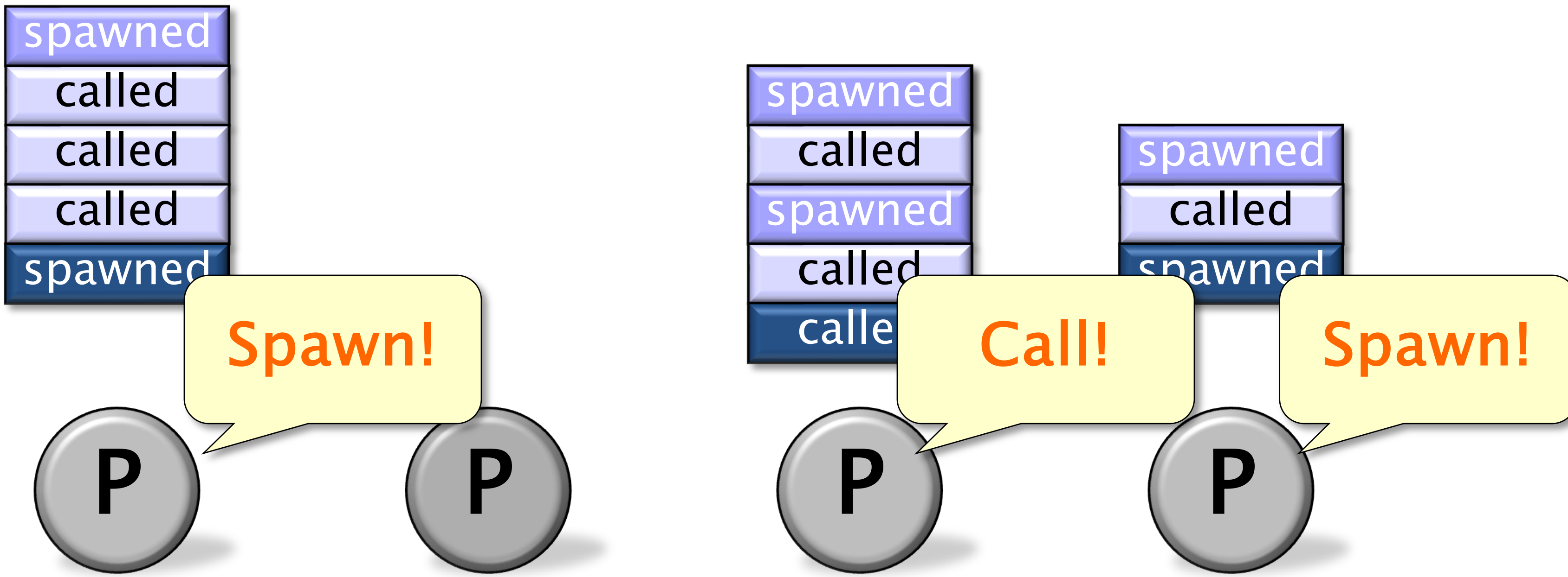
Work Stealing

Each worker (processor) maintains a work deque of ready strands, and it manipulates the bottom of the deque like a stack [MKH90, BL94, FLR98].



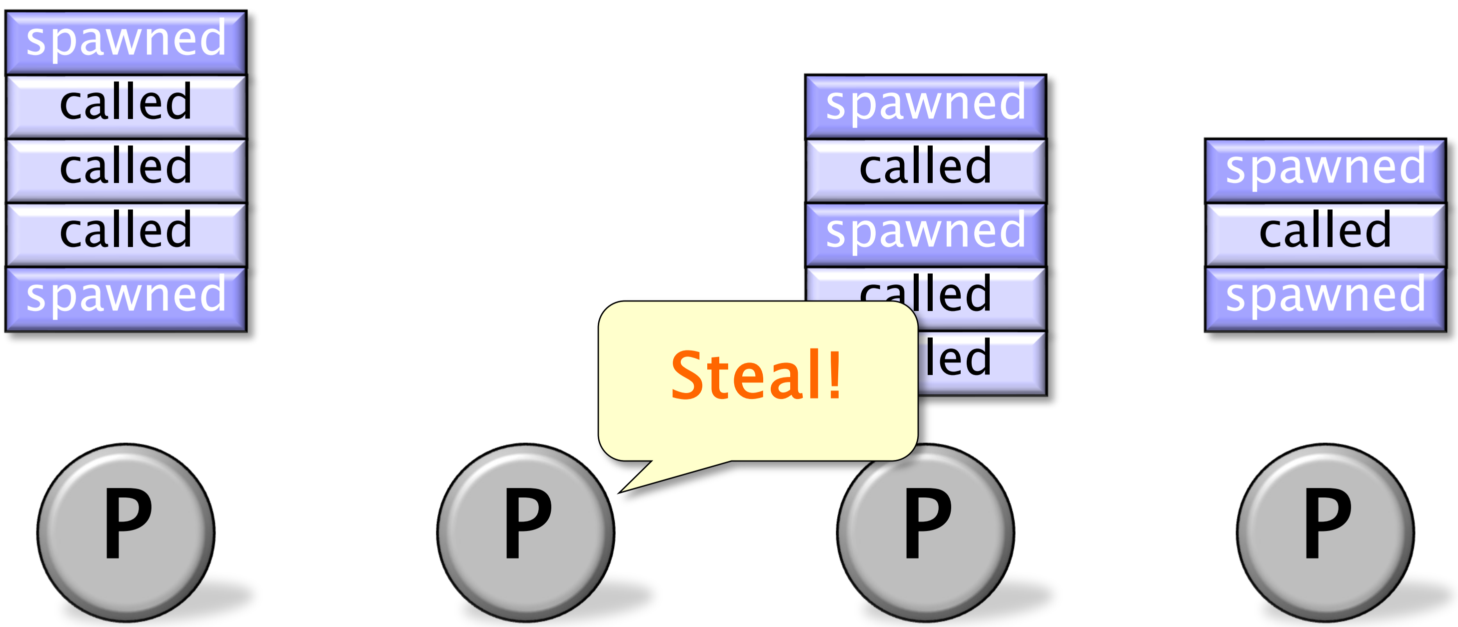
Work Stealing

Each worker (processor) maintains a work deque of ready strands, and it manipulates the bottom of the deque like a stack [MKH90, BL94, FLR98].



Work Stealing

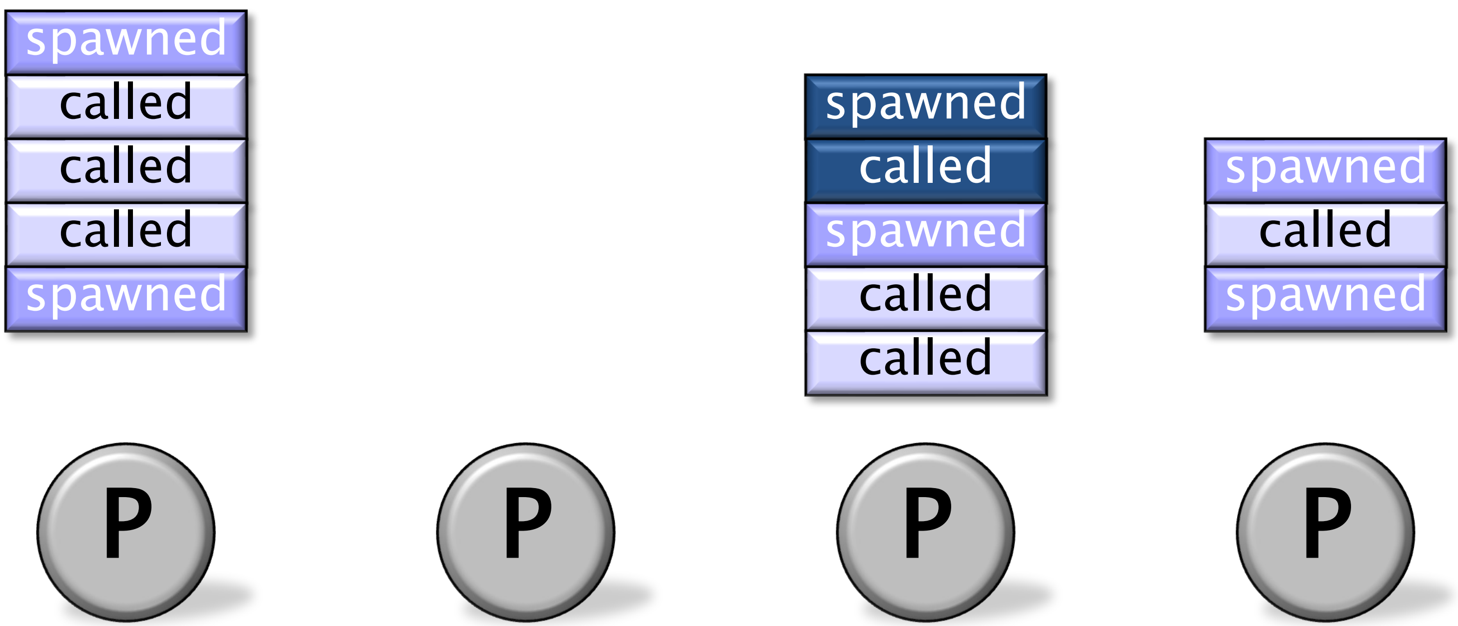
Each worker (processor) maintains a work deque of ready strands, and it manipulates the bottom of the deque like a stack [MKH90, BL94, FLR98].



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

Work Stealing

Each worker (processor) maintains a work deque of ready strands, and it manipulates the bottom of the deque like a stack [MKH90, BL94, FLR98].



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

Work-Stealing Bounds

The performance of a Cilk program depends on two measures:

- *Work* , T_1 — total executed instructions
- *Span*, T_∞ — length of a longest path of serial dependencies

Theorem [BL94]. OpenCilk's randomized work-stealing scheduler achieves expected running time

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

on P processors.

T_P is within a constant factor of optimal.

Pseudoproof of Work-Stealing Bounds

Theorem [BL94]. OpenCilk's randomized work-stealing scheduler achieves expected running time

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

on P processors.

Pseudoproof. 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 cost of all steals is $O(PT_\infty)$. Since there are P processors, the expected time is

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

What Do These Bounds Mean?

Theorem [BL94]. OpenCilk's randomized work-stealing scheduler achieves expected running time

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

on P processors.

Time workers spend working.

Time workers spend stealing.

If the program achieves linear speedup, then workers spend most of their time working.

Scalability vs. Speedup

Ideally, parallelization should make a serial code run P times faster on P processors.

Serial matrix multiply

```
for (int i = 0; i < n; ++i)
  for (int k = 0; k < n; ++k)
    for (int j = 0; j < n; ++j)
      C[i][j] += A[i][k] * B[k][j];
```

Running time T_S.

Cilk matrix multiply

```
cilk_for (int i = 0; i < n; ++i)
  for (int k = 0; k < n; ++k)
    for (int j = 0; j < n; ++j)
      C[i][j] += A[i][k] * B[k][j];
```

With sufficient parallelism, running time $T_P \approx T_1/P$.

Goal: $T_P \approx T_S/P$, meaning that $T_S \approx T_1$.

Work Efficiency

Consider a Cilk program, and define:
T_1 — work of the Cilk program
T_∞ — span of the Cilk program
T_S — work of an analogous serial code

To achieve linear speedup on P processors over its serial analogue — i.e., $T_P \approx T_S/P$ — the parallel program must exhibit:

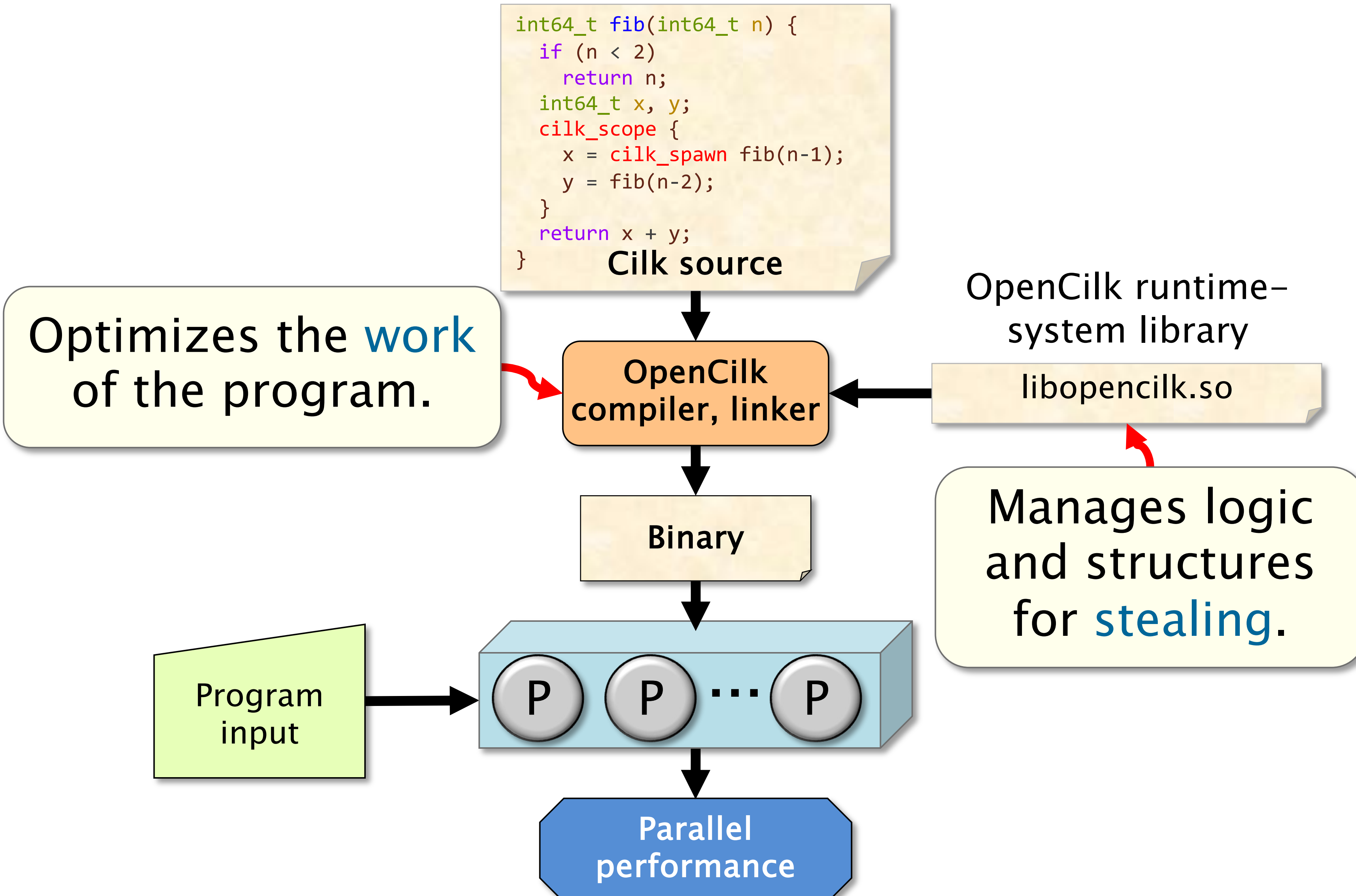
- Ample parallelism: $T_1/T_\infty \gg P$.
- High work efficiency: $T_S/T_1 \approx 1$.

The Work-First Principle

To optimize the execution of programs with sufficient parallelism, the implementation of OpenCilk follows the **work-first principle**:

> Optimize for the *ordinary serial execution*, at the expense of some additional computation in steals.

OpenCilk Platform



Version 6: Parallel Loops

Version	Implementation	Running time (s)	Relative speedup	Absolute Speedup	GFLOPS	Percent of peak
1	Python	21041.67	1.00	1	0.006	0.001
2	Java	2387.32	8.81	9	0.058	0.007
3	C	1155.77	2.07	18	0.118	0.014
4	+ interchange loops	177.68	6.50	118	0.774	0.093
5	+ optimization flags	54.63	3.25	385	2.516	0.301
6	Parallel loops	3.04	17.97	6,921	45.211	5.408

Parallelizing the `i` loop yields a speedup of almost 18× on 18 cores!

- Disclaimer: It's rarely this easy to parallelize code effectively. Most code requires far more creativity to achieve a good speedup.

Let's try this on SpeedCode!

Version 6: Parallel Loops

Version	Implementation	Running time (s)	Relative speedup	Absolute Speedup	GFLOPS	Percent of peak
1	Python	21041.67	1.00	1	0.006	0.001
2	Java	2387.32	8.81	9	0.058	0.007
3	C	1155.77	2.07	18	0.118	0.014
4	+ interchange loops	177.68	6.50	118	0.774	0.093
5	+ optimization flags	54.63	3.25	385	2.516	0.301
6	Parallel loops	3.04	17.97	6,921	45.211	5.408

Parallelizing the `i` loop yields a speedup of almost 18× on 18 cores!

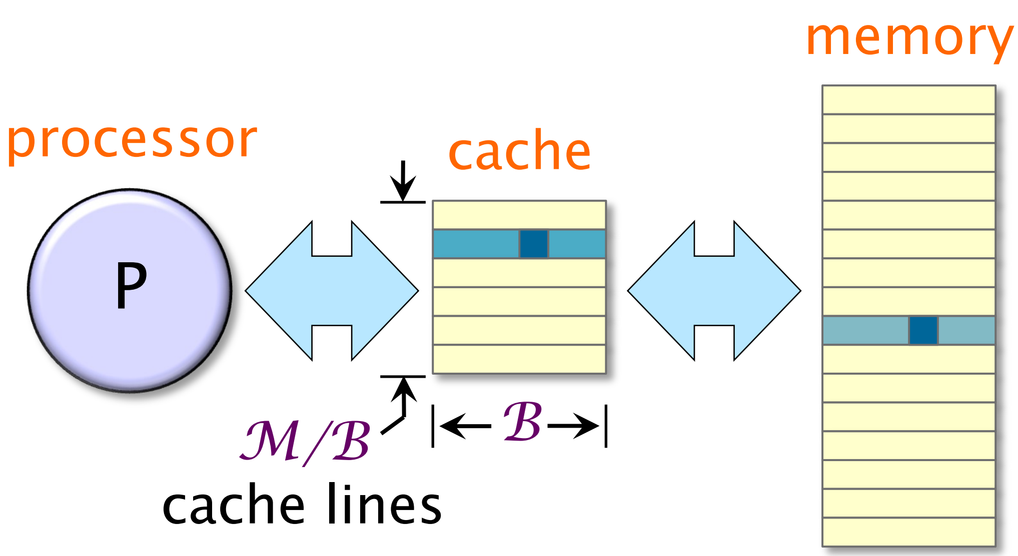
- Disclaimer: It's rarely this easy to parallelize code effectively. Most code requires far more creativity to achieve a good speedup.

Why are we still getting barely 5% of peak?

Hardware Caches, Revisited

IDEA: Restructure the computation to reuse data in the cache as much as possible.

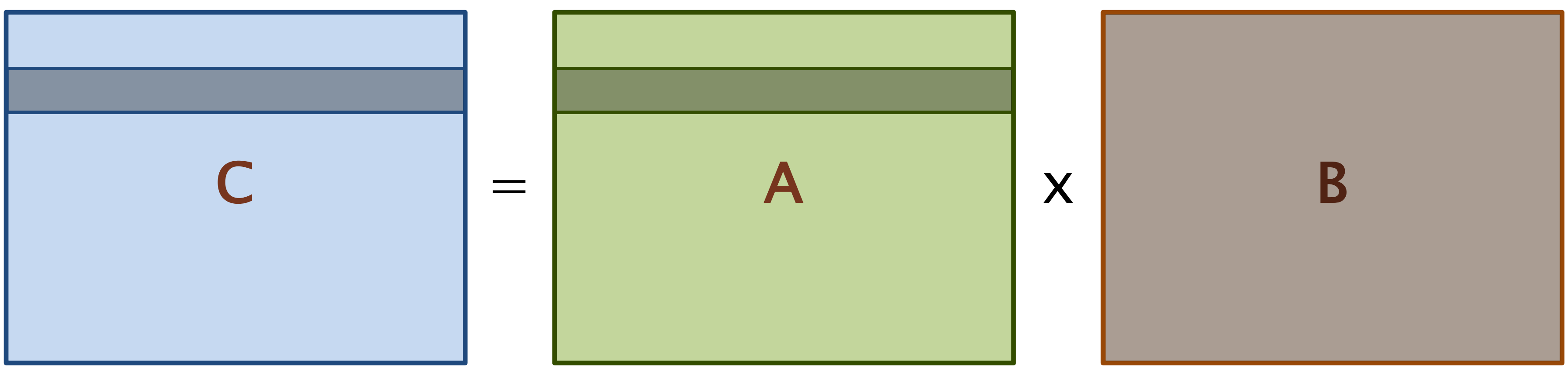
- Cache misses are slow, and cache hits are fast.
- Try to make the most of the cache by reusing the data that's already there.



Data Reuse: Loops

How many memory accesses must the looping code perform to fully compute 1 row of C?

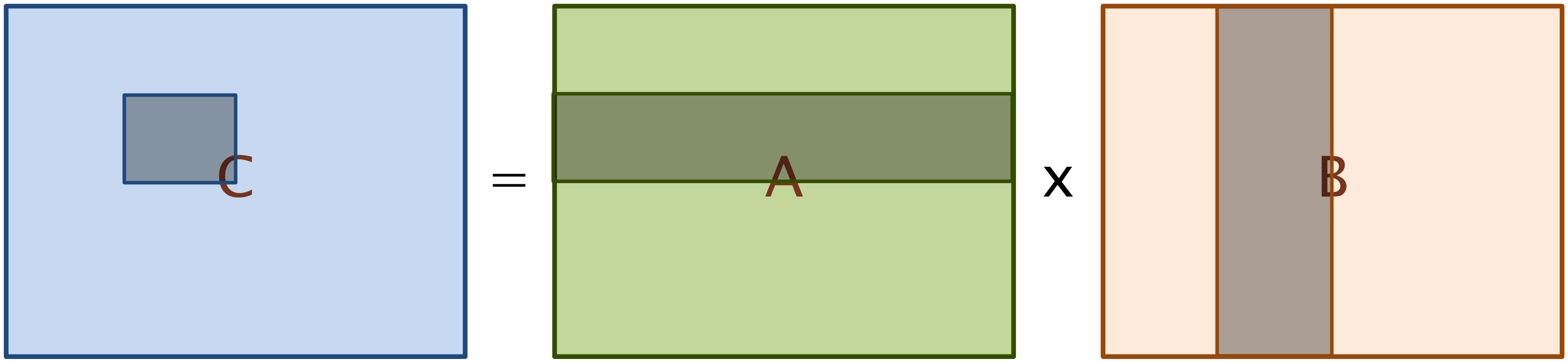
- 4096 * 1 = 4096 writes to C,
- 4096 * 1 = 4096 reads from A, and
- 4096 * 4096 = 16,777,216 reads from B, which is
- 16,785,408 memory accesses total.



Data Reuse: Blocks

How about to compute a 64×64 block of C?

- $64 \cdot 64 = 4096$ writes to C,
- $64 \cdot 4096 = 262{,}144$ reads from A, and
- $4096 \cdot 64 = 262{,}144$ reads from B, or
- $528{,}384$ memory accesses total.



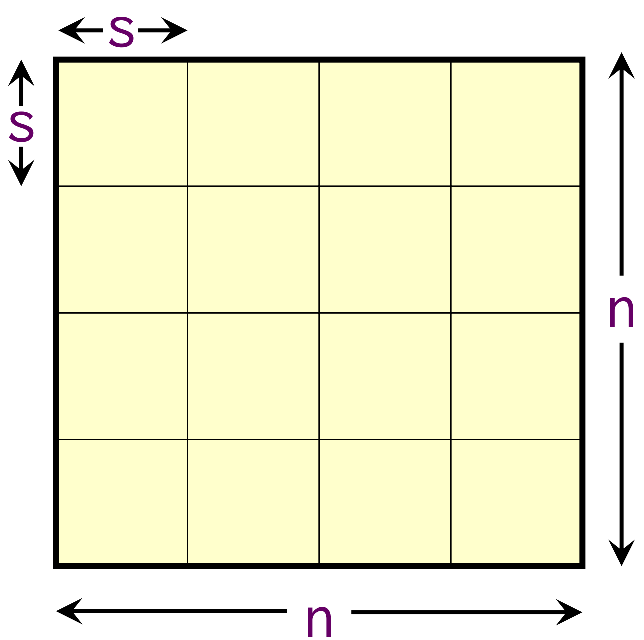
Tiled Matrix Multiplication

```
cilk_for (int ih = 0; ih < n; ih += s)
  cilk_for (int jh = 0; jh < n; jh += s)
    for (int kh = 0; kh < n; kh += s)
      for (int il = 0; il < s; ++il)
        for (int kl = 0; kl < s; ++kl)
          for (int jl = 0; jl < s; ++jl)
            C[ih+il][jh+jl] += A[ih+il][kh+kl] * B[kh+kl][jh+jl];
```

Tiled Matrix Multiplication

```
cilk_for (int ih = 0; ih < n; ih += s)
  cilk_for (int jh = 0; jh < n; jh += s)
    for (int kh = 0; kh < n; kh += s)
      for (int il = 0; il < s; ++il)
        for (int kl = 0; kl < s; ++kl)
          for (int jl = 0; jl < s; ++jl)
            C[ih+il][jh+jl] += A[ih+il][kh+kl] * B[kh+kl][jh+jl];
```

Tuning parameter
How do we find the right value of s?
Experiment!



Tile size	Running time (s)
4	6.74
8	2.76
16	2.49
32	**1.74**
64	2.33
128	2.13

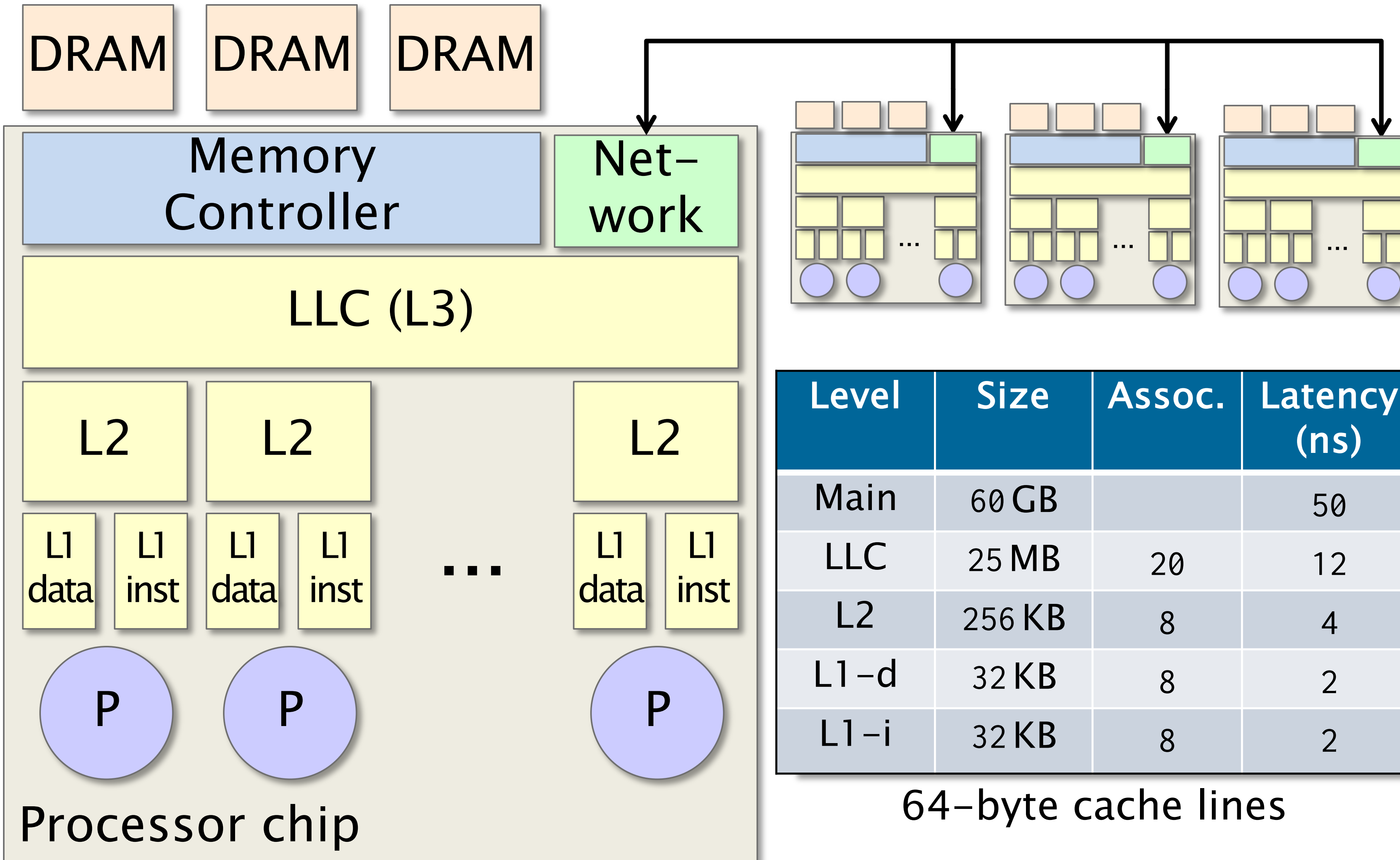
Tiling Performance

Version	Implementation	Running time (s)	Relative speedup	Absolute Speedup	GFLOPS	Percent of peak
1	Python	21041.67	1.00	1	0.006	0.001
2	Java	2387.32	8.81	9	0.058	0.007
3	C	1155.77	2.07	18	0.118	0.014
4	+ interchange loops	177.68	6.50	118	0.774	0.093
5	+ optimization flags	54.63	3.25	385	2.516	0.301
6	Parallel loops	3.04	17.97	6,921	45.211	5.408
	+ tiling	1.79	1.70	11,772	76.782	9.184

Implementation	Cache references $\times 10^6$	L1-d cache misses $\times 10^6$	Last-level cache misses $\times 10^6$
Parallel loops	104,090	17,220	8,600
+ tiling	64,690	11,777	416

The tiled implementation performs about 40% fewer cache references and 95% fewer last-level cache misses.

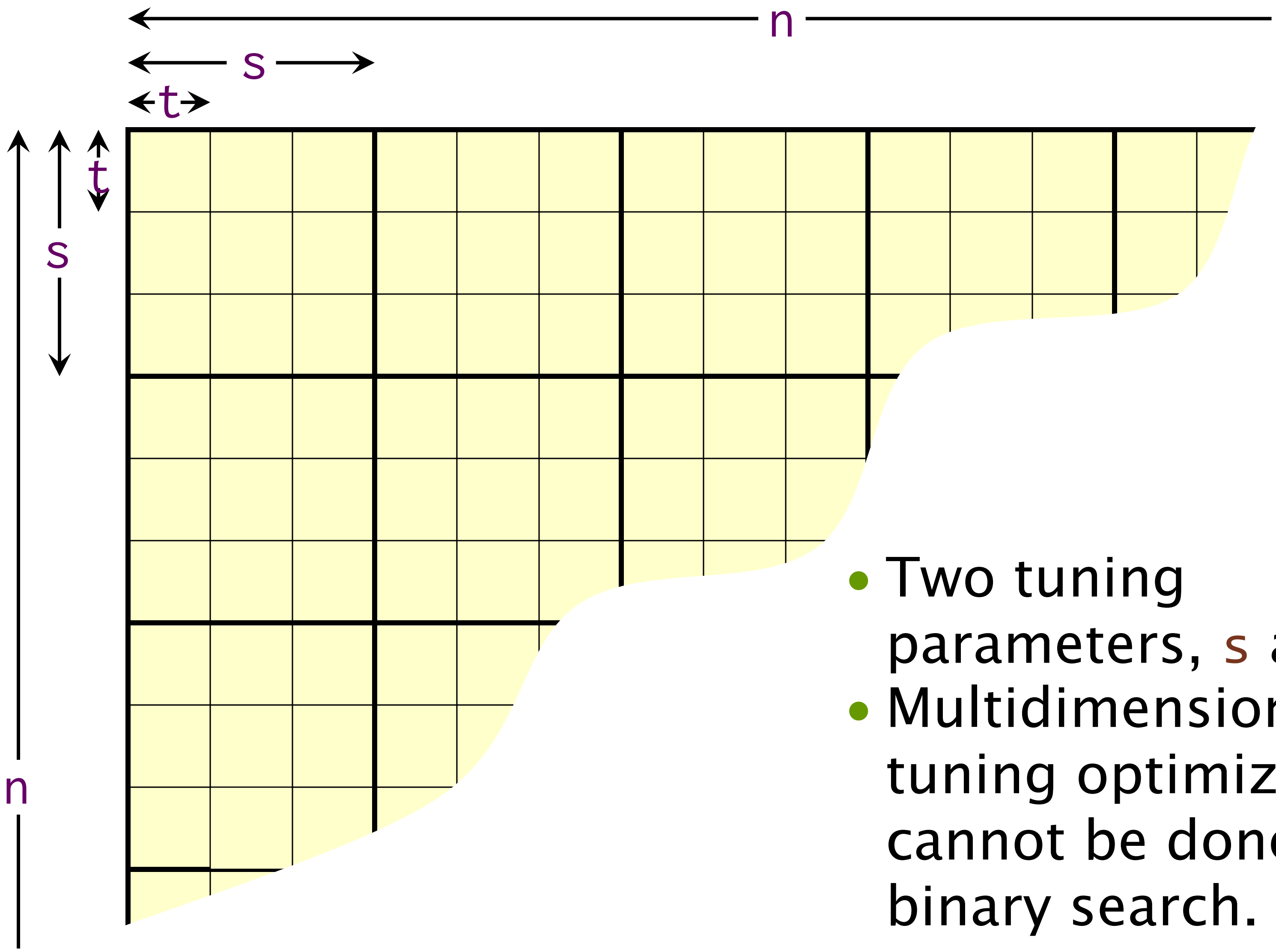
Multicore Cache Hierarchy



Level	Size	Assoc.	Latency (ns)
Main	60 GB		50
LLC	25 MB	20	12
L2	256 KB	8	4
L1-d	32 KB	8	2
L1-i	32 KB	8	2

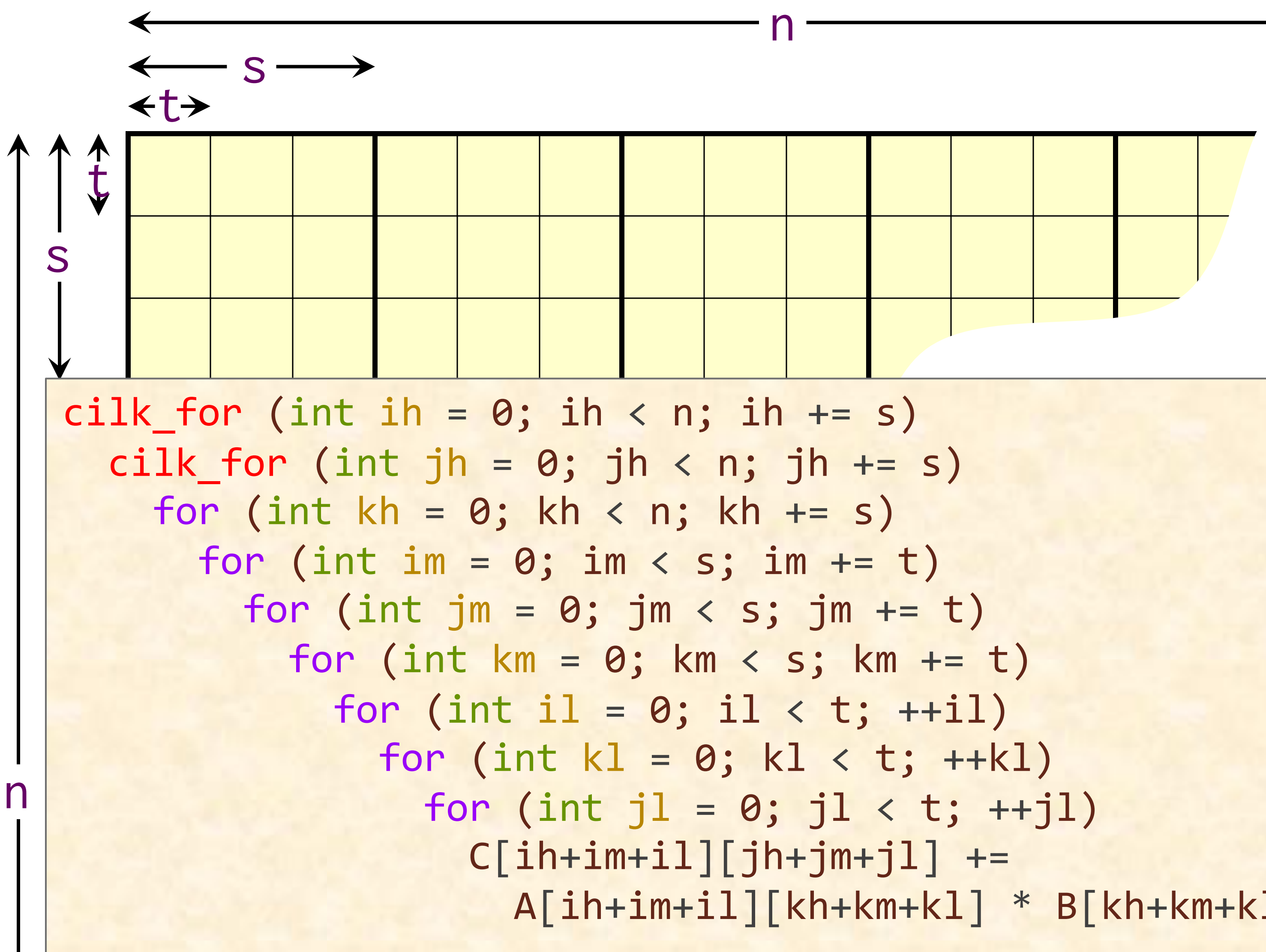
64-byte cache lines

Tiling for a Two-Level Cache



- Two tuning parameters, `s` and `t`.
- Multidimensional tuning optimization cannot be done with binary search.

Tiling for a Two-Level Cache




```
cilk_for (int ih = 0; ih < n; ih += s)
  cilk_for (int jh = 0; jh < n; jh += s)
    for (int kh = 0; kh < n; kh += s)
      for (int im = 0; im < s; im += t)
        for (int jm = 0; jm < s; jm += t)
          for (int km = 0; km < s; km += t)
            for (int il = 0; il < t; ++il)
              for (int kl = 0; kl < t; ++kl)
                for (int jl = 0; jl < t; ++jl)
                  C[ih+im+il][jh+jm+jl] +=
                    A[ih+im+il][kh+km+kl] * B[kh+km+kl][jh+jm+jl];
```

D&C Matrix Multiplication

For matrix multiplication, a recursive, parallel, divide-and-conquer algorithm uses caches almost optimally.

$$\begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix} = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} \cdot \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix}$$

IDEA: Divide the matrices into $(n/2)\times(n/2)$ submatrices.

D&C Matrix Multiplication

For matrix multiplication, a recursive, parallel, divide-and-conquer algorithm uses caches almost optimally.

$$\begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix} = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} \cdot \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix}$$

$$= \begin{bmatrix} A_{00}B_{00} & A_{00}B_{01} \\ A_{10}B_{00} & A_{10}B_{01} \end{bmatrix} + \begin{bmatrix} A_{01}B_{10} & A_{01}B_{11} \\ A_{11}B_{10} & A_{11}B_{11} \end{bmatrix}$$

1. Compute $C_{00} \mathrel{+}= A_{00}B_{00}$; $C_{01} \mathrel{+}= A_{00}B_{01}$; $C_{10} \mathrel{+}= A_{10}B_{00}$; and $C_{11} \mathrel{+}= A_{10}B_{01}$ recursively in parallel.
2. Compute $C_{00} \mathrel{+}= A_{01}B_{10}$; $C_{01} \mathrel{+}= A_{01}B_{11}$; $C_{10} \mathrel{+}= A_{11}B_{10}$; and $C_{11} \mathrel{+}= A_{11}B_{11}$ recursively in parallel.

Recursive Parallel Matrix Multiply

```c
void mm_dac(double *restrict C, int n_C,
            double *restrict A, int n_A,
            double *restrict B, int n_B,
            int n)
{ // C += A * B
  assert((n & (-n)) == n);
  if (n <= THRESHOLD) {
    mm_base(C, n_C, A, n_A, B, n_B
  } else {
#define X(M,row,col) (M    ow*(n_ ## M) + col)*(n/2))
    cilk_scope {
      cilk_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
      cilk_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
      cilk_spawn mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
      cilk_spawn mm_dac(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
    }
    cilk_scope {
      cilk_spawn mm_dac(X(C,0,
      cilk_spawn mm_dac(X(C,0,1), n_C, X(A,
      cilk_spawn mm_dac(X(C,1,0), n_C, X(A,1,1
      cilk_spawn mm_dac(X(C,1,1), n_C, X(A,1,1
} } }
```

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

Control cannot exit this scope until all spawned children have returned.

Recursive Parallel Matrix Multiply

```
void mm_dac(double *restrict C, int n_C,
            double *restrict A, int n_A,
            double *restrict B, int n_B,
            int n)
{ // C += A * B
  assert((n & (-n)) == n);
  if (n <= THRESHOLD) {
    mm_base(C, n_C, A, n_A, B, n_B, n);
  } else {
#define X(M,row,col) (M + (row*(n_ ## M) + col)*(n/2))
    cilk_scope {
      cilk_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
      cilk_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
      cilk_spawn mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
      cilk_spawn mm_dac(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
    }
    cilk_scope {
      cilk_spawn mm_dac(X(C,0,0), n_C, X(A,0,1), n_A, X(B,1,0), n_B, n/2);
      cilk_spawn mm_dac(X(C,0,1), n_C, X(A,0,1), n_A, X(B,1,1), n_B, n/2);
      cilk_spawn mm_dac(X(C,1,0), n_C, X(A,1,1), n_A, X(B,1,0), n_B, n/2);
      cilk_spawn mm_dac(X(C,1,1), n_C, X(A,1
} } }
```

Do 4 subproblems in parallel…

…and when they're done, do the other 4.

Version 7: Parallel Divide-and-Conquer

Version	Implementation	Running time (s)	Relative speedup	Absolute Speedup	GFLOPS	Percent of peak
1	Python	21041.67	1.00	1	0.006	0.001
2	Java	2387.32	8.81	9	0.058	0.007
3	C	1155.77	2.07	18	0.118	0.014
4	+ interchange loops	177.68	6.50	118	0.774	0.093
5	+ optimization flags	54.63	3.25	385	2.516	0.301
6	Parallel loops	3.04	17.97	6,921	45.211	5.408
7	Parallel divide-and-conquer	1.30	2.35	16,197	105.722	12.646

Implementation	Cache references $\times 10^6$	Cache references $\times 10^6$	L1-d cache misses $\times 10^6$
Parallel loops	104,090	17,220	8,600
+ tiling	64,690	11,777	416
Parallel divide-and-conquer	58,230	9,407	64

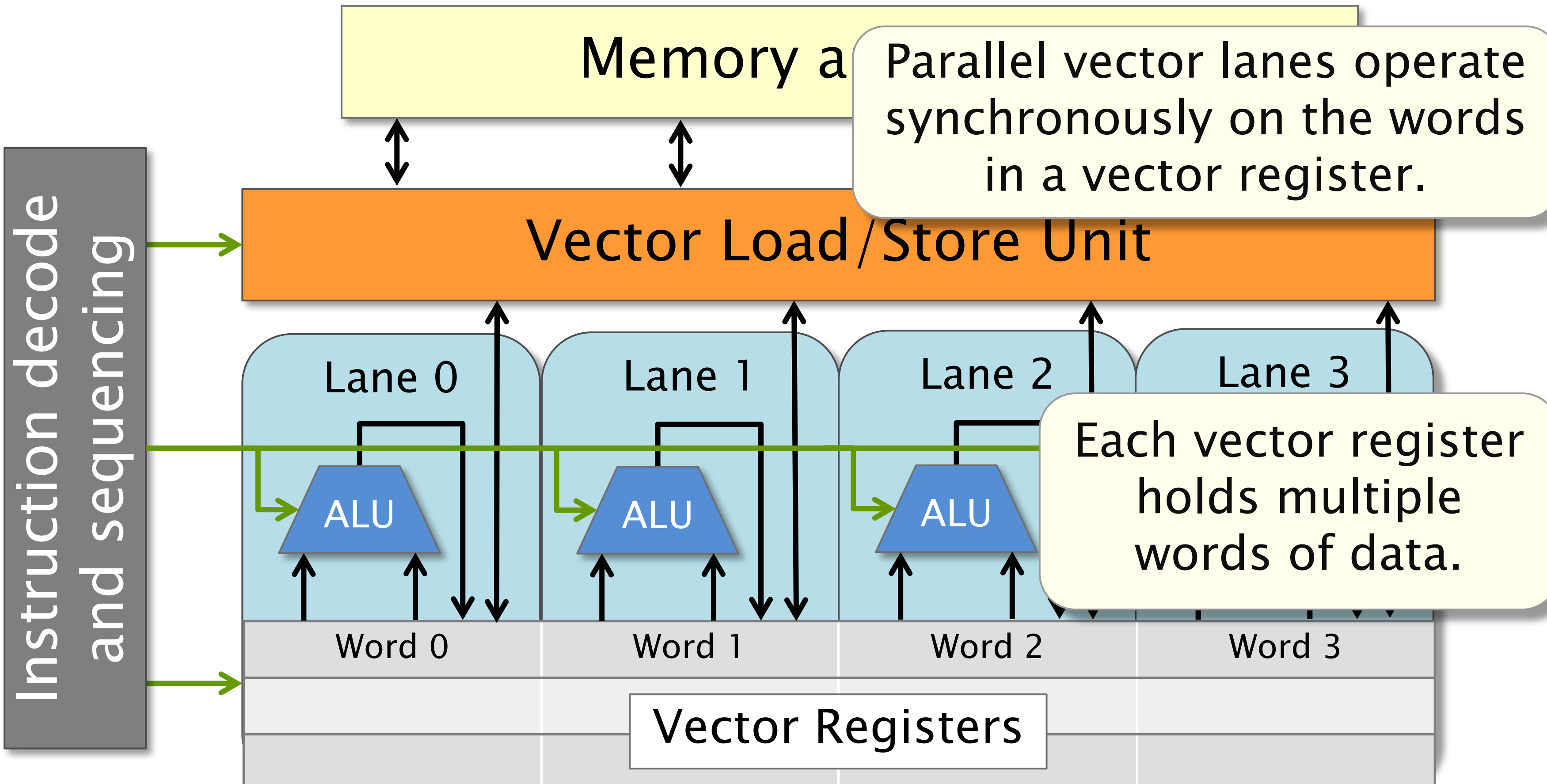
Version 7: Parallel Divide-and-Conquer

Version	Implementation	Running time (s)	Relative speedup	Absolute Speedup	GFLOPS	Percent of peak
1	Python	21041.67	1.00	1	0.006	0.001
2	Java	2387.32	8.81	9	0.058	0.007
3	C	1155.77	2.07	18	0.118	0.014
4	+ interchange loops	177.68	6.50	118	0.774	0.093
5	+ optimization flags	54.63	3.25	385	2.516	0.301
6	Parallel loops	3.04	17.97	6,921	45.211	5.408
7	Parallel divide-and-conquer	1.30	2.35	16,197	105.722	12.646

Challenge: Performance-engineer this algorithm on SpeedCode!

Vector Hardware

Modern microprocessors incorporate vector hardware to process data in single-instruction stream, multiple-data stream (SIMD) fashion.



Compiler Vectorization

Clang/LLVM uses vector instructions automatically when compiling at optimization level -O2 or higher.

Clang/LLVM can be induced to produce a ***vectorization report*** as follows:

```
$ clang -O3 -std=c99 mm.c -o mm -Rpass=vector
mm.c:42:7: remark: vectorized loop (vectorization width: 2,
interleaved count: 2) [-Rpass=loop-vectorize]
      for (int j = 0; j < n; ++j) {
      ^
```

Many machines don't support the newest set of vector instructions, however, so the compiler uses vector instructions conservatively by default.

Version 8: Compiler Vectorization

Version	Implementation	Running time (s)	Relative speedup	Absolute Speedup	GFLOPS	Percent of peak
1	Python	21041.67	1.00	1	0.006	0.001
2	Java	2387.32	8.81	9	0.058	0.007
3	C	1155.77	2.07	18	0.118	0.014
4	+ interchange loops	177.68	6.50	118	0.774	0.093
5	+ optimization flags	54.63	3.25	385	2.516	0.301
6	Parallel loops	3.04	17.97	6,921	45.211	5.408
7	Parallel divide-and-conquer	1.30	2.35	16,197	105.722	12.646
8	+ compiler vectorization	0.70	1.87	30,272	196.341	23.486

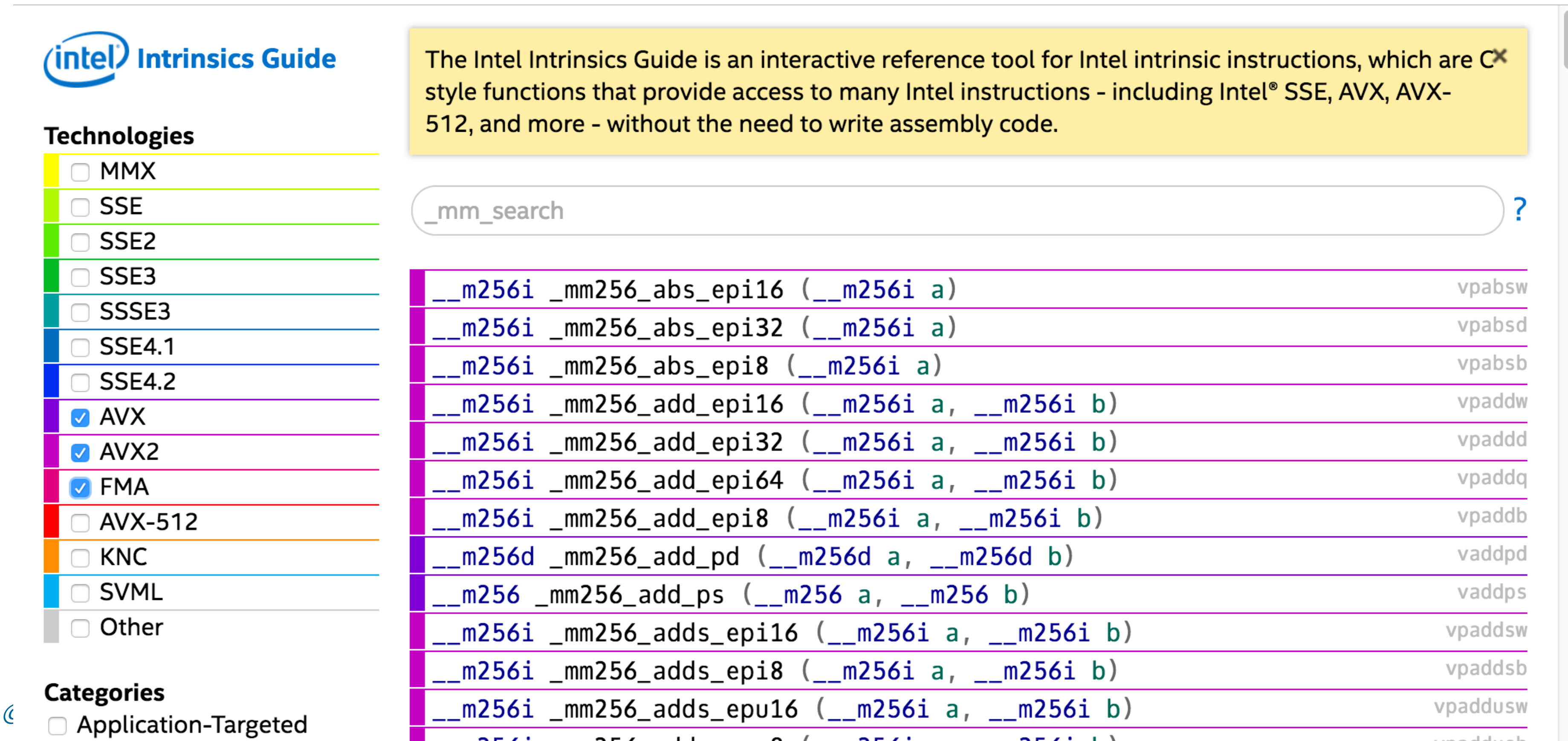
Using the flag `-march=native` nearly doubles the program's performance!

Can we be smarter than the compiler?

AVX Intrinsic Instructions

Intel provides C-style functions, called *intrinsic instructions*, that provide direct access to hardware vector operations:

https://software.intel.com/sites/landingpage/IntrinsicsGuide/



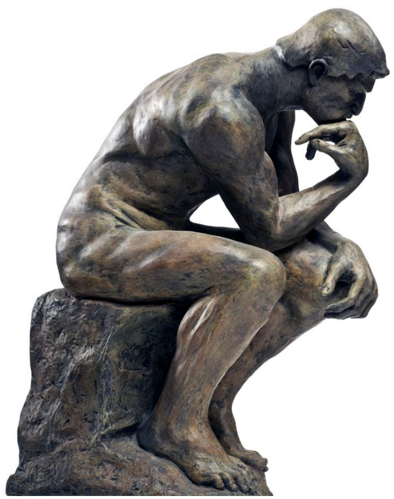
Plus More Optimizations

We can apply several more insights and performance-engineering tricks to make this code run faster, including:

- Preprocessing
- Matrix transposition
- Data layout
- Memory-management optimizations
- A clever algorithm for the base case that manages vector registers and instructions explicitly

Plus Performance Engineering

Think,



code,

run, run, run…

…to test and measure many different implementations

Version 9: AVX Intrinsics

Version	Implementation	Running time (s)	Relative speedup	Absolute Speedup	GFLOPS	Percent of peak
1	Python	21041.67	1.00	1	0.006	0.001
2	Java	2387.32	8.81	9	0.058	0.007
3	C	1155.77	2.07	18	0.118	0.014
4	+ interchange loops	177.68	6.50	118	0.774	0.093
5	+ optimization flags	54.63	3.25	385	2.516	0.301
6	Parallel loops	3.04	17.97	6,921	45.211	5.408
7	Parallel divide-and-conquer	1.30	2.35	16,197	105.722	12.646
8	+ compiler vectorization	0.70	1.87	30,272	196.341	23.486
9	+ AVX intrinsics	0.39	1.76	53,292	352.408	41.677

Version 10: Final Reckoning

Version	Implementation	Running time (s)	Relative speedup	Absolute Speedup	GFLOPS	Percent of peak
1	Python	21041.67	1.00	1	0.006	0.001
2	Java	2387.32	8.81	9	0.058	0.007
3	C	1155.77	2.07	18	0.118	0.014
4	+ interchange loops	177.68	6.50	118	0.774	0.093
5	+ optimization flags	54.63	3.25	385	2.516	0.301
6	Parallel loops	3.04	17.97	6,921	45.211	5.408
7	Parallel divide-and-conquer	1.30	2.35	16,197	105.722	12.646
8	+ compiler vectorization	0.70	1.87	30,272	196.341	23.486
9	+ AVX intrinsics	0.39	1.76	53,292	352.408	41.677
10	Intel MKL	0.41	0.97	51,497	335.217	40.098

Our Version 9 is competitive with Intel's professionally engineered Math Kernel Library!

Performance Engineering

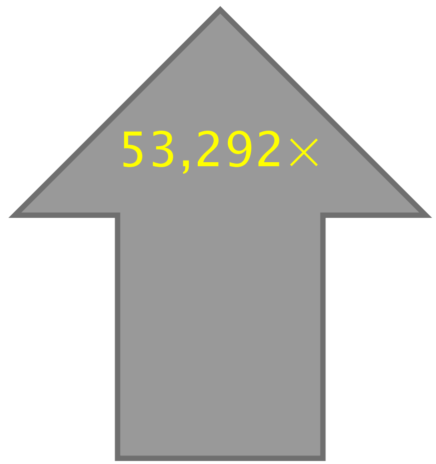
- You won't generally see the magnitude of performance improvement we obtained for matrix multiplication.

Galopagos
Tortoise
0.5 k/h

Performance Engineering

- You won’t generally see the magnitude of performance improvement we obtained for matrix multiplication.

Escape Velocity
11 k/s

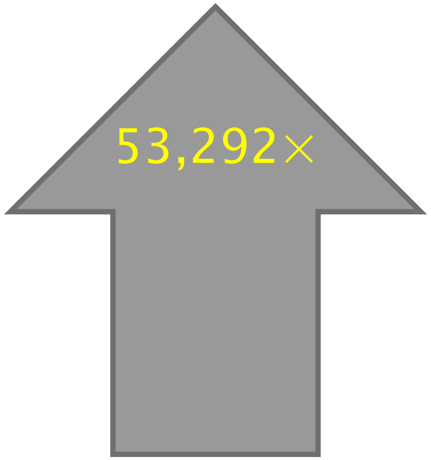


Galopagos Tortoise
0.5 k/h

Performance Engineering

- You won't generally see the magnitude of performance improvement we obtained for matrix multiplication.
- But 6.106 will teach you how to print the currency of performance all by yourself.

Escape Velocity
11 k/s



Galopagos Tortoise
0.5 k/h

QUESTIONS?