Parallelism and programming languages

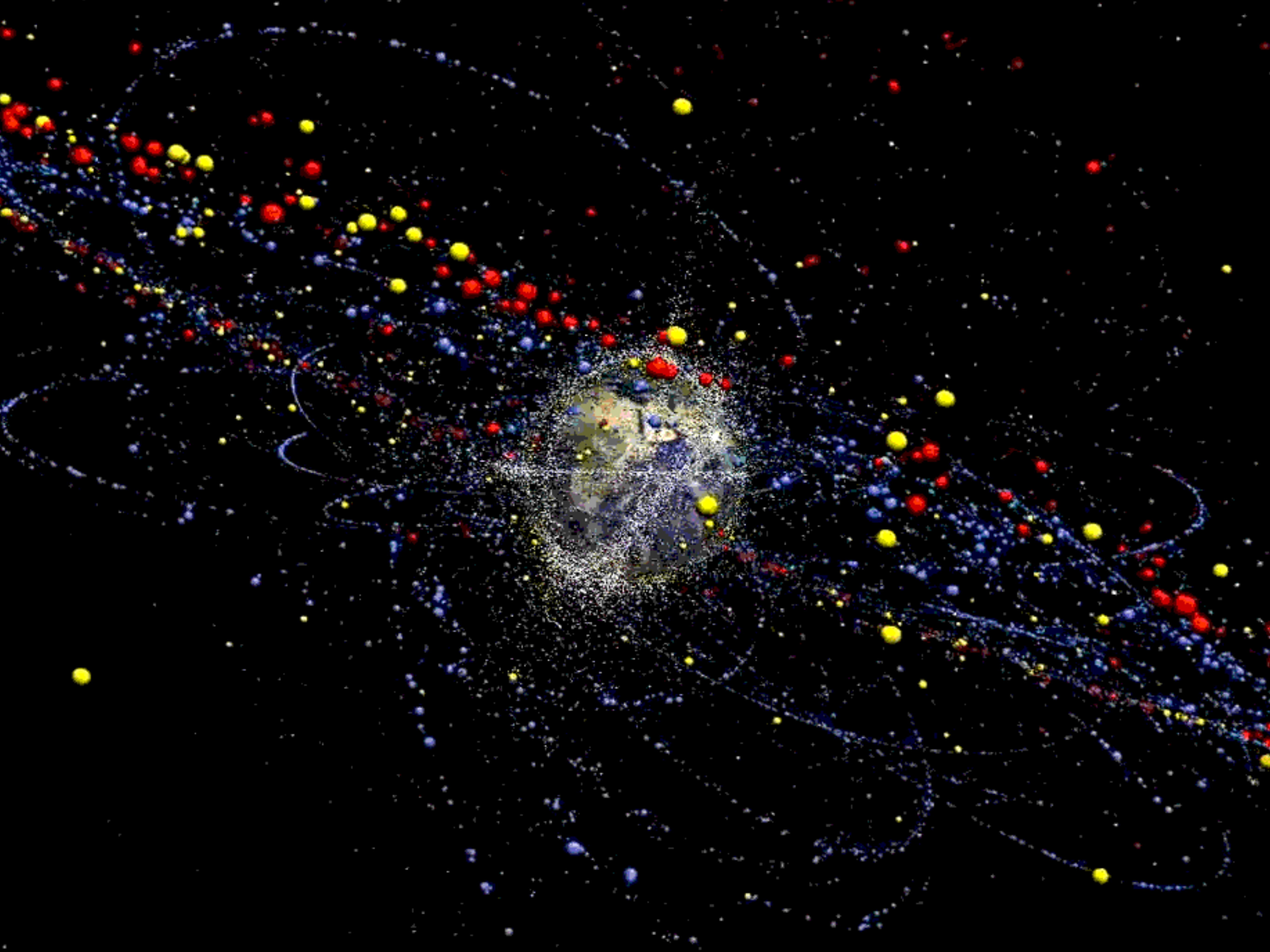
CS 242 – 11/20/19

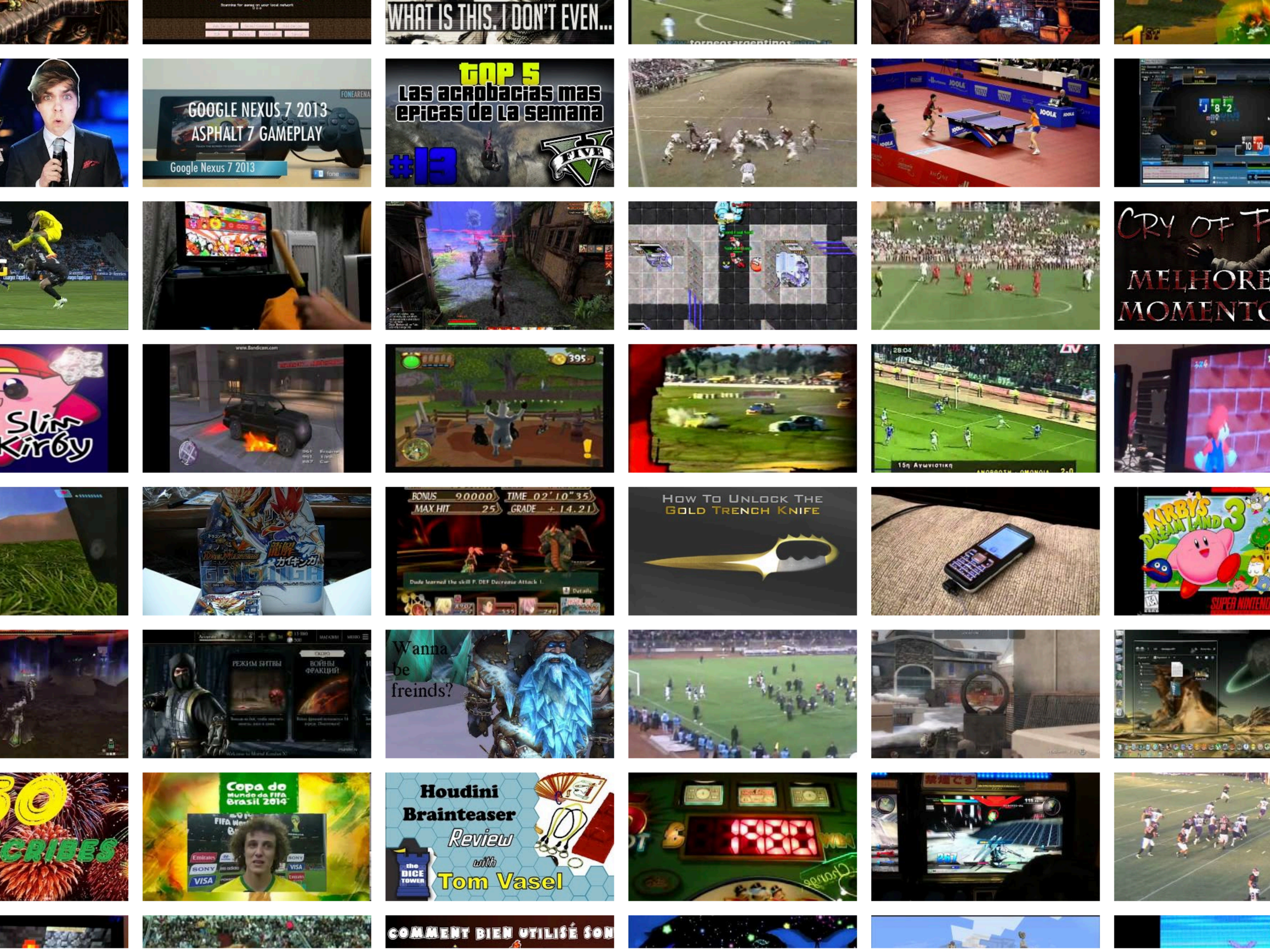
Concurrency vs. Parallelism

Competitive | *Cooperative*

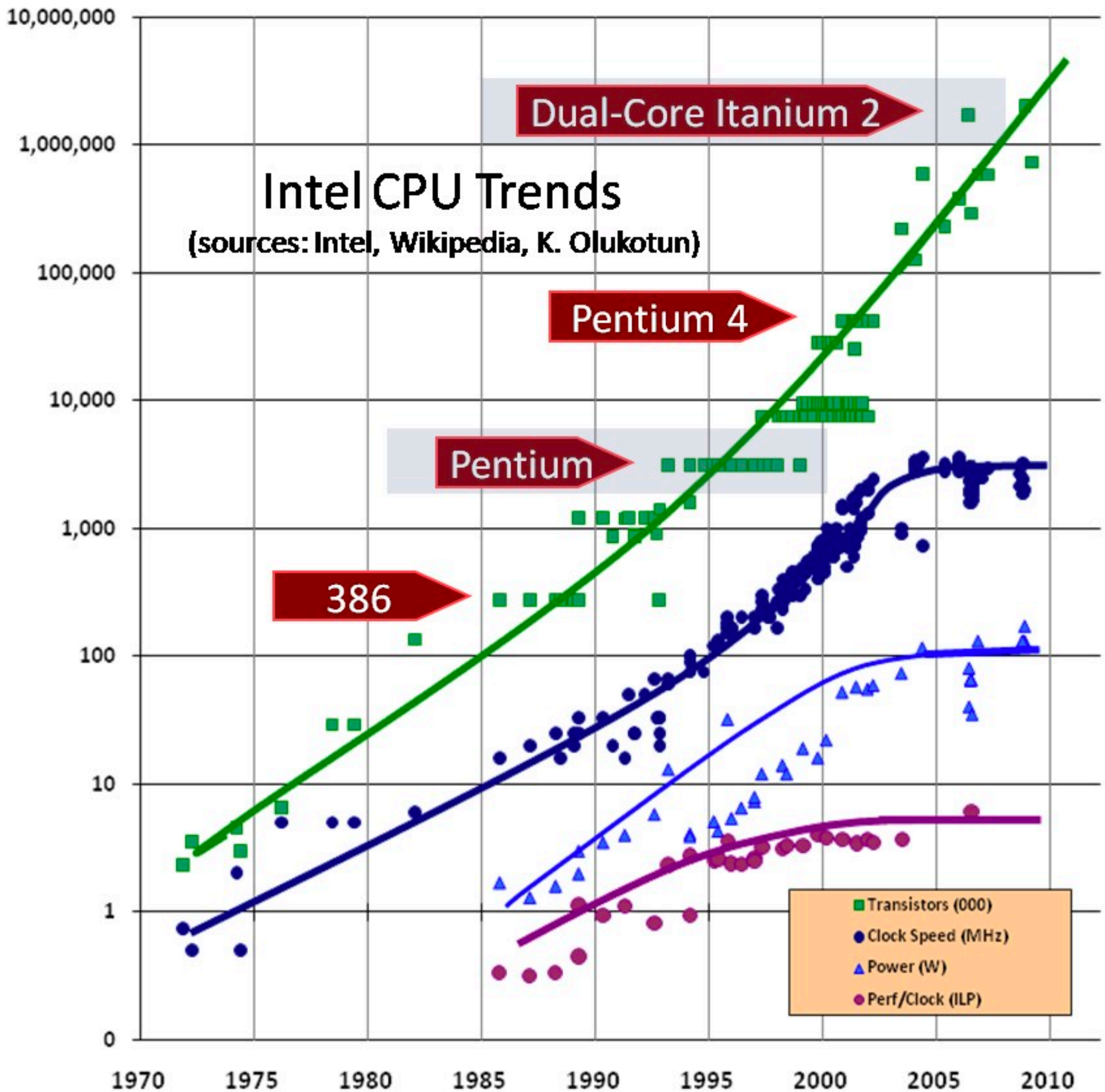
Parallelism:

Use multiple resources to accomplish a goal faster.

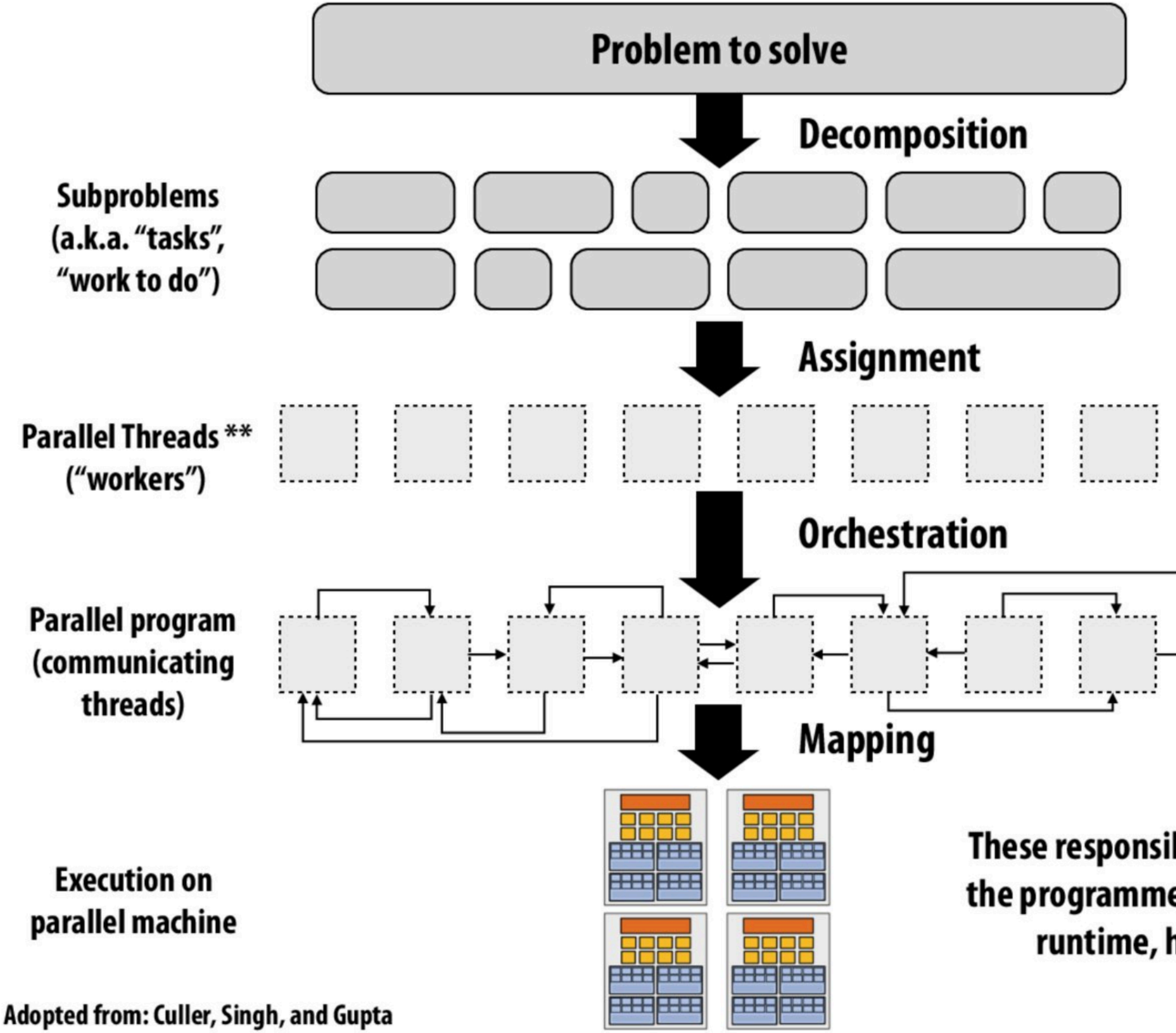



WHAT IS THIS, I DON'T EVEN...
FONEARENA
GOOGLE NEXUS 7 2013
ASPHALT 7 GAMEPLAY
Google Nexus 7 2013
TOP 5
las acrobacias mas
epicas de la semana
#13
FIVE
MELHORE
www.Bandicam.com
395
28:04
BONUS 90000
MAX HIT 25
TIME 02'10"35
GRADE +14.21
Dude learned the skill P. DEF Decrease Attack I.
How To Unlock The
Gold Trench Knife
KIRBY'S DREAM LAND 3
РЕЖИМ БИТВЫ
ВОЙНЫ ФРАКЦИЙ
Wanna
be
freinds?
Copa do
Mundo da FIFA
Brasil 2014
Houdini
Brainteaser
Review
with
Tom Vasel
the
DICE
TOWER
COMMENT BIEN UTILISÉ SON

Single-core is tapped out (mostly)



Creating a parallel program



** I had to pick a term

These responsibilities may be assumed by the programmer, by the system (compiler, runtime, hardware), or by both!

Adopted from: Culler, Singh, and Gupta

Sum prime numbers in a vector

```rust
fn main() {
  let vec: Vec<i64> = (0..100000).collect();

  // Imperative version
  let mut sum: i64 = 0;
  for i in vec.iter() {
    if is_prime(i) {
      sum += i;
    }
  }

  // Functional version
  let sum: i64 = vec.iter().filter(is_prime).sum();

  println!("Sum: {}", sum);
}
```

Sum prime numbers in a vector in parallel

```rust
use std::{thread, sync::Arc};
const NUM_WORKERS: usize = 8;

fn main() {
  let vec: Arc<Vec<i64>> = Arc::new((0..100000).collect());

  let chunk_size: usize = vec.len() / NUM_WORKERS;

  let handles: Vec<thread::JoinHandle<i64>> =
    (0..NUM_WORKERS).map(|i| {
      let vec_ref = vec.clone();
      thread::spawn(move || {
        let range = (i * chunk_size) .. ((i + 1) * chunk_size);
        vec_ref[range].iter().filter(is_prime).sum()
      })
    }).collect();

  let mut final_sum = 0;
  for handle in handles {
    final_sum += handle.join().unwrap();
  }

  println!("Sum: {}", final_sum);
}
```

1. Decomposition: K chunks

2. Assignment

4. Mapping

3. Orchestration

How can we achieve the same effect without all the extra code?

Idea #1: use same language, and try to find parallelism

```
let mut sum: i64 = 0;
for i in 0..100000 {
  if is_prime(i) {
    sum += i;
  }
}
```

Compiler Magic

```
let handles: =
 (0..NUM_WORKERS).map(|i| {
  let vec_ref = vec.clone();
  thread::spawn(move || {
    ..
  })
 }).collect();
```

Register allocation with graph coloring

			live-in	
f_1	$\leftarrow$	1	$\cdot$	
f_2	$\leftarrow$	1	f_1	
f_3	$\leftarrow$	$f_2 + f_1$	f_2, f_1	
f_4	$\leftarrow$	$f_3 + f_2$	f_3, f_2	
f_5	$\leftarrow$	$f_4 + f_3$	f_4, f_3	
`%eax`	$\leftarrow$	f_5	f_5	
return			`%eax`	return register

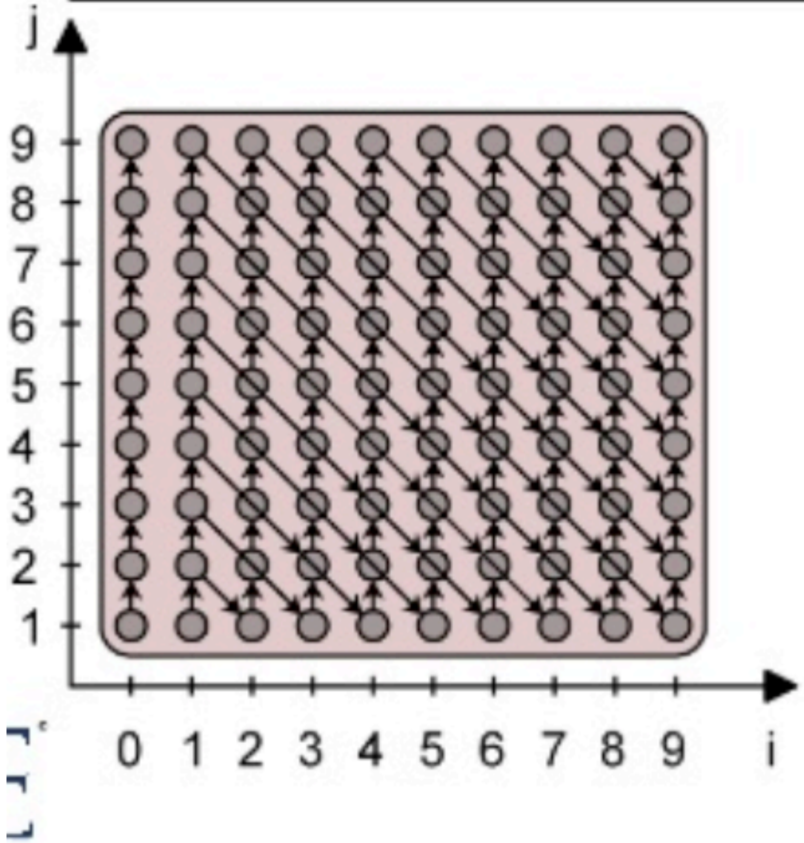
$$\begin{array}{ccccccc} 0 & & 1 & & 0 & & 1 \\ f_1 & — & f_2 & — & f_3 & — & f_4 \end{array}$$

`%eax`	$\leftarrow$	1
`%edx`	$\leftarrow$	1
`%eax`	$\leftarrow$	`%edx` + `%eax`
`%edx`	$\leftarrow$	`%eax` + `%edx`
`%eax`	$\leftarrow$	`%edx` + `%eax`
`%eax`	$\leftarrow$	`%eax`

Frank Pfenning, CMU 15-411

Polyhedral analysis for auto-parallelization

```
for (i = 0; i < N; i++) {
  for (j = 1; j < N; j++) {
    a[i][j] = a[j][i] + a[i][j-1]; // S1
  }
}
```




```
a[0][1] = a[1][0] + a[0][0]; // S1(0,1)
a[0][2] = a[2][0] + a[0][1]; // S1(0,2)
a[0][3] = a[3][0] + a[0][2]; // S1(0,3)
...
a[1][1] = a[1][1] + a[1][0]; // S1(1,1)
a[1][2] = a[2][1] + a[1][1]; // S1(1,2)
a[1][3] = a[3][1] + a[1][2]; // S1(1,3)
...
a[2][1] = a[1][2] + a[2][0]; // S1(2,1)
a[2][2] = a[2][2] + a[2][1]; // S1(2,2)
a[2][3] = a[3][2] + a[2][2]; // S1(2,3)
...
a[3][1] = a[1][3] + a[3][0]; // S1(3,1)
```

Akihiro Hayashi and Jun Shirako, "Introduction to Polyhedral Compilation" 2016

"Autovectorization is not a programming model"

For a long time most of the Intel compiler team denied that anything more than their auto-vectorizer was needed to take care of vector unit utilization. We quickly fell into a cycle:

- They'd inform the graphics folks that they'd improved their auto-vectorizer in response to our requests and that it did everything we had asked for.
- We'd try it and find that though it was better, boy was it easy to write code that wasn't actually compiled to vector code—it'd fail unpredictably.
- We'd give them failing cases, a few months would would pass and they'd inform us that the latest version solved the problem.

It didn't take much to fall off the vectorization path. They tried to patch things up at first, but eventually came up with `#pragma simd`, which would disable the "is it safe to vectorize this" checks in the auto-vectorizer and vectorize the following loop no matter what.

Matt Pharr, "The story of ispc: origins (part 1)" 2018

Idea #2: restrict the language to make parallelism implicit

Rayon library in Rust

```
use rayon::prelude::*;

fn main() {
  let vec: Vec<i64> = (0..100000).collect();
  let sum: i64 = vec.par_iter().filter(is_prime).sum();
  println!("Sum: {}", sum);
}
```

Canonical list processing operations

Transformations	$map(f : \mathrm{T} \Rightarrow \mathrm{U})$	:	RDD[T] ⇒ RDD[U]
	$filter(f : \mathrm{T} \Rightarrow \mathrm{Bool})$	:	RDD[T] ⇒ RDD[T]
	$flatMap(f : \mathrm{T} \Rightarrow \mathrm{Seq[U]})$	:	RDD[T] ⇒ RDD[U]
	$sample(fraction : \mathrm{Float})$	:	RDD[T] ⇒ RDD[T] (Deterministic sampling)
	$groupByKey()$	:	RDD[(K, V)] ⇒ RDD[(K, Seq[V])]
	$reduceByKey(f : (\mathrm{V}, \mathrm{V}) \Rightarrow \mathrm{V})$	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	$union()$	:	(RDD[T], RDD[T]) ⇒ RDD[T]
	$join()$	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (V, W))]
	$cogroup()$	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (Seq[V], Seq[W]))]
	$crossProduct()$	:	(RDD[T], RDD[U]) ⇒ RDD[(T, U)]
	$mapValues(f : \mathrm{V} \Rightarrow \mathrm{W})$	:	RDD[(K, V)] ⇒ RDD[(K, W)] (Preserves partitioning)
	$sort(c : \mathrm{Comparator[K]})$	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	$partitionBy(p : \mathrm{Partitioner[K]})$	:	RDD[(K, V)] ⇒ RDD[(K, V)]
Actions	$count()$	:	RDD[T] ⇒ Long
	$collect()$	:	RDD[T] ⇒ Seq[T]
	$reduce(f : (\mathrm{T}, \mathrm{T}) \Rightarrow \mathrm{T})$	:	RDD[T] ⇒ T
	$lookup(k : \mathrm{K})$	:	RDD[(K, V)] ⇒ Seq[V] (On hash/range partitioned RDDs)
	$save(path : \mathrm{String})$	:	Outputs RDD to a storage system, *e.g.,* HDFS

Table 2: Transformations and actions available on RDDs in Spark. Seq[T] denotes a sequence of elements of type T.

Zaharia et al. "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing." NSDI'12

Theoretical parallelism is well-understood

Operation	Work	Span
length a	1	1
nth a i	1	1
singleton x	1	1
empty	1	1
isSingleton x	1	1
isEmpty x	1	1
tabulate f n	$1 + \sum_{i=0}^{n} W(f(i))$	$1 + \max_{i=0}^{n} S(f(i))$
map f a	$1 + \sum_{x \in a} W(f(x))$	$1 + \max_{x \in a} S(f(x))$
filter f a	$1 + \sum_{x \in a} W(f(x))$	$\lg\lvert a\rvert + \max_{x \in a} S(f(x))$
subseq a (i, j)	1	1
append a b	$1 + \lvert a\rvert + \lvert b\rvert$	1
flatten a	$1 + \lvert a\rvert + \sum_{x \in a} \lvert x\rvert$	$1 + \lg\lvert a\rvert$
update a (i, x)	$1 + \lvert a\rvert$	1
inject a b	$1 + \lvert a\rvert + \lvert b\rvert$	1
collect f a	$1 + W(f) \cdot \lvert a\rvert \lg\lvert a\rvert$	$1 + S(f) \cdot \lg^2\lvert a\rvert$
iterate f x a	$1 + \sum_{f(y,z) \in \mathcal{T}(-)} W(f(y,z))$	$1 + \sum_{f(y,z) \in \mathcal{T}(-)} S(f(y,z))$
reduce f x a	$1 + \sum_{f(y,z) \in \mathcal{T}(-)} W(f(y,z))$	$\lg\lvert a\rvert \cdot \max_{f(y,z) \in \mathcal{T}(-)} S(f(y,z))$

Umut Acar and Guy Blelloch. "Algorithms: Parallel and Sequential." 2019

Hardest part of parallelism isn't expressing parallelism

```
use rayon::prelude::*;

fn main() {
  let vec: Vec<i64> = (0..100000).collect();
  let sum: i64 = vec.par_iter().filter(is_prime).sum();
  println!("Sum: {}", sum);
}
```

Work imbalance!

Redistribute work evenly

```
vec.par_iter().filter(|n| is_prime(n)).shuffle().sum();
```

Scheduling parallelism for image processing kernel in Halide

Blurring an image in C++

(a) Clean C++ : 9.94 ms per megapixel

```
void blur(const Image &in, Image &blurred) {
 Image tmp(in.width(), in.height());

 for (int y = 0; y < in.height(); y++)
  for (int x = 0; x < in.width(); x++)
   tmp(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;

 for (int y = 0; y < in.height(); y++)
  for (int x = 0; x < in.width(); x++)
   blurred(x, y) = (tmp(x, y-1) + tmp(x, y) + tmp(x, y+1))/3;
}
```

Ragan-Kelley et al. “Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines.” SIGGRAPH 2012

Blurring an image *quickly* in C++

(b) Fast C++ (for x86) : 0.90 ms per megapixel

```
void fast_blur(const Image &in, Image &blurred) {
 __m128i one_third = _mm_set1_epi16(21846);
 #pragma omp parallel for
 for (int yTile = 0; yTile < in.height(); yTile += 32) {
  __m128i a, b, c, sum, avg;
  __m128i tmp[(256/8)*(32+2)];
  for (int xTile = 0; xTile < in.width(); xTile += 256) {
   __m128i *tmpPtr = tmp;
   for (int y = -1; y < 32+1; y++) {
    const uint16_t *inPtr = &(in(xTile, yTile+y));
    for (int x = 0; x < 256; x += 8) {
     a = _mm_loadu_si128((__m128i*)(inPtr-1));
     b = _mm_loadu_si128((__m128i*)(inPtr+1));
     c = _mm_load_si128((__m128i*)(inPtr));
     sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
     avg = _mm_mulhi_epi16(sum, one_third);
     _mm_store_si128(tmpPtr++, avg);
     inPtr += 8;
   }}
   tmpPtr = tmp;
   for (int y = 0; y < 32; y++) {
    __m128i *outPtr = (__m128i *)(&(blurred(xTile, yTile+y)));
    for (int x = 0; x < 256; x += 8) {
     a = _mm_load_si128(tmpPtr+(2*256)/8);
     b = _mm_load_si128(tmpPtr+256/8);
     c = _mm_load_si128(tmpPtr++);
```

For image processing, the global organization of execution and storage is critical. Image processing pipelines are both wide and deep: they consist of many data-parallel stages that benefit hugely from parallel execution across pixels, but stages are often memory bandwidth limited—they do little work per load and store.

Gains in speed therefore come not just from optimizing the inner loops, but also from global program transformations such as tiling and fusion that exploit producer-consumer locality down the pipeline. The best choice of transformations is architecture-specific; implementations optimized for an x86 multicore and for a modern GPU often bear little resemblance to each other.

Ragan-Kelley et al. "Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines." SIGGRAPH 2012