Java™


ORACLE

# Divide and Conquer Parallelism with the Fork/Join Framework

Mark Reinhold (@mreinhold)
*Chief Architect, Java Platform Group*

2011/7/7

10,000000

1,000,000

100,000

10,000

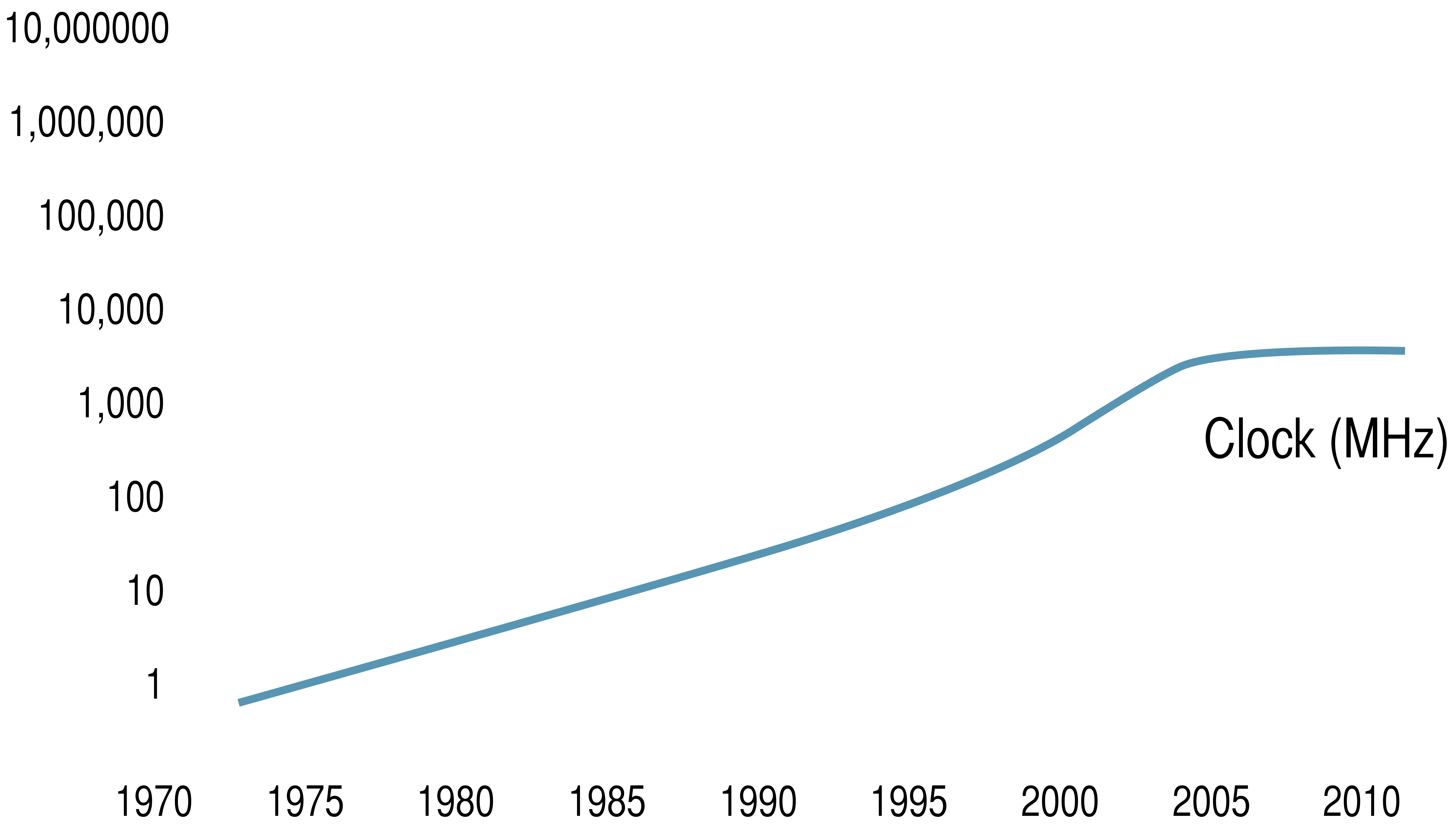
1,000

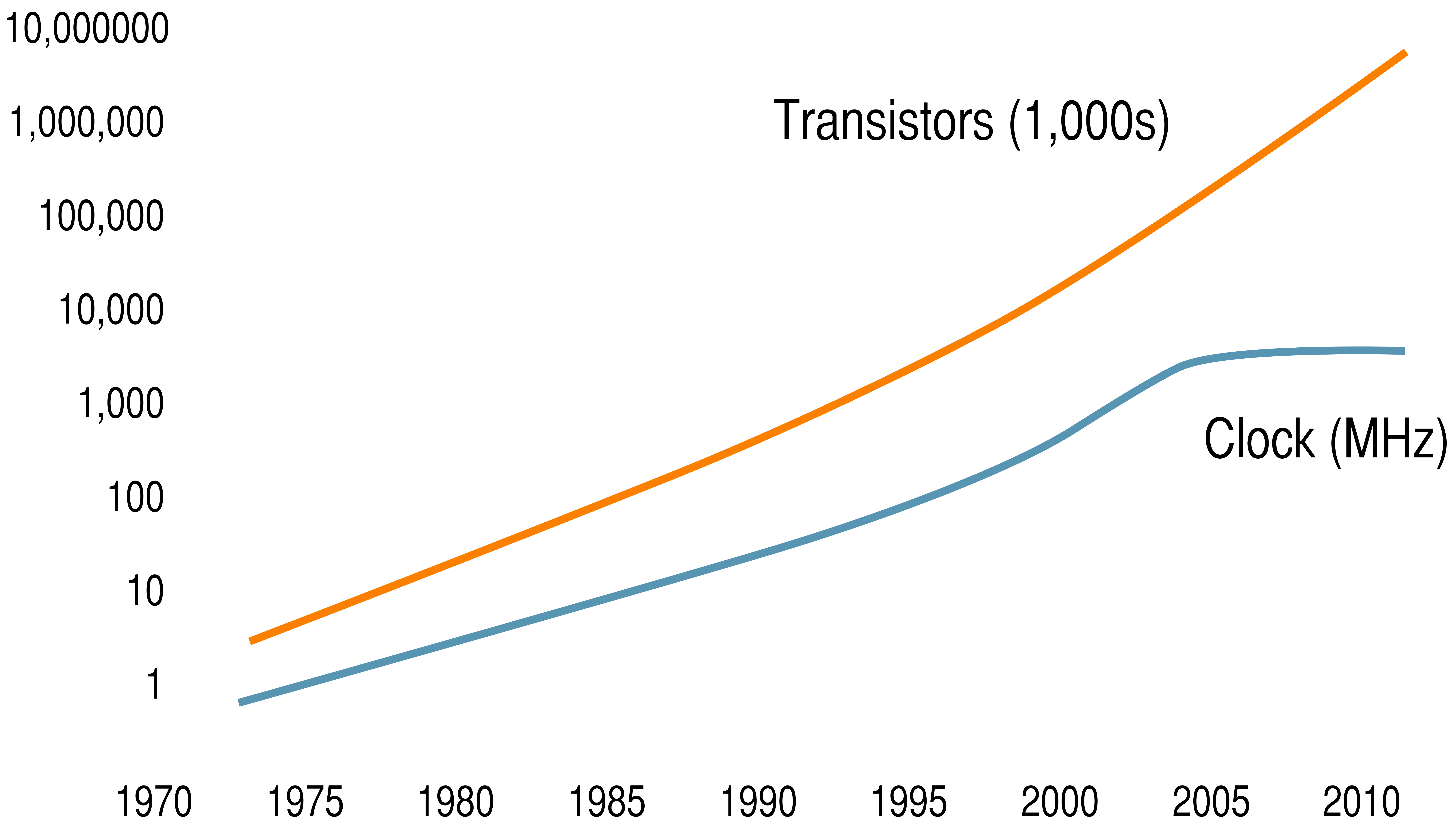
100

10

1

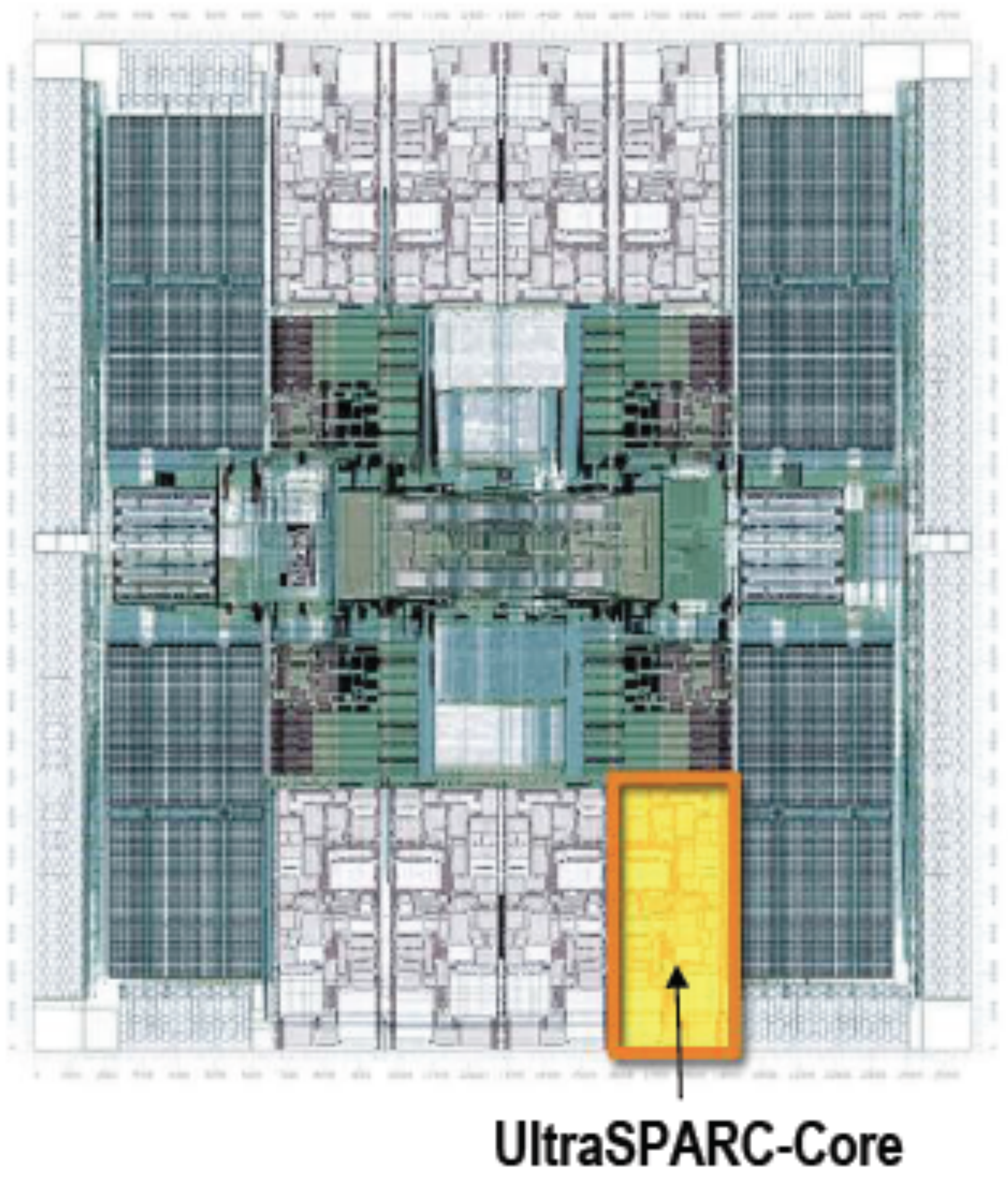
1970 1975 1980 1985 1990 1995 2000 2005 2010


10,000000
1,000,000
100,000
10,000
1,000
100
10
1
Clock (MHz)
1970
1975
1980
1985
1990
1995
2000
2005
2010



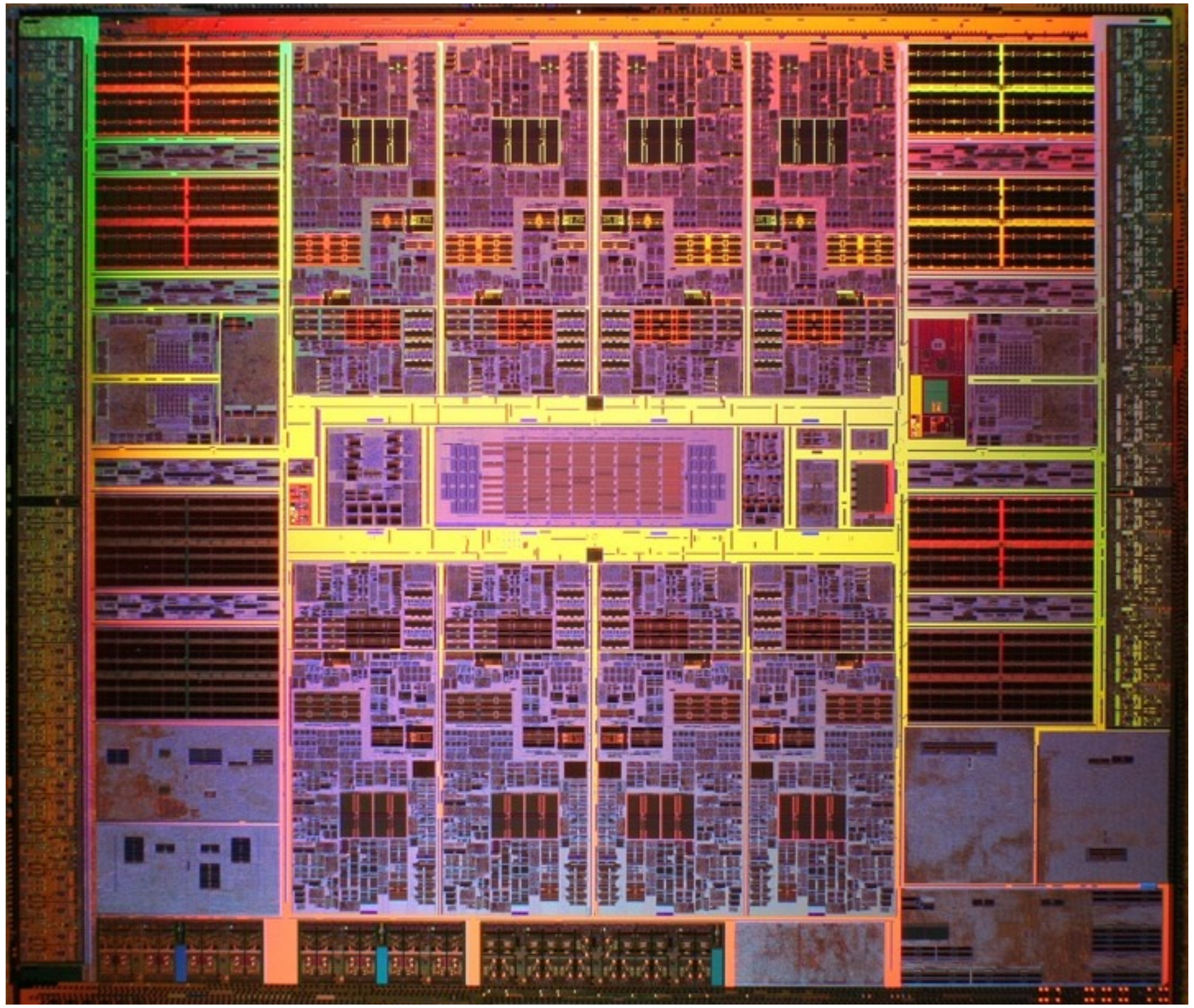
10,000000
1,000,000
100,000
10,000
1,000
100
10
1
Transistors (1,000s)
Clock (MHz)
1970
1975
1980
1985
1990
1995
2000
2005
2010

ORACLE®




# Niagara 1 (2005)

8 x 4 = **32**

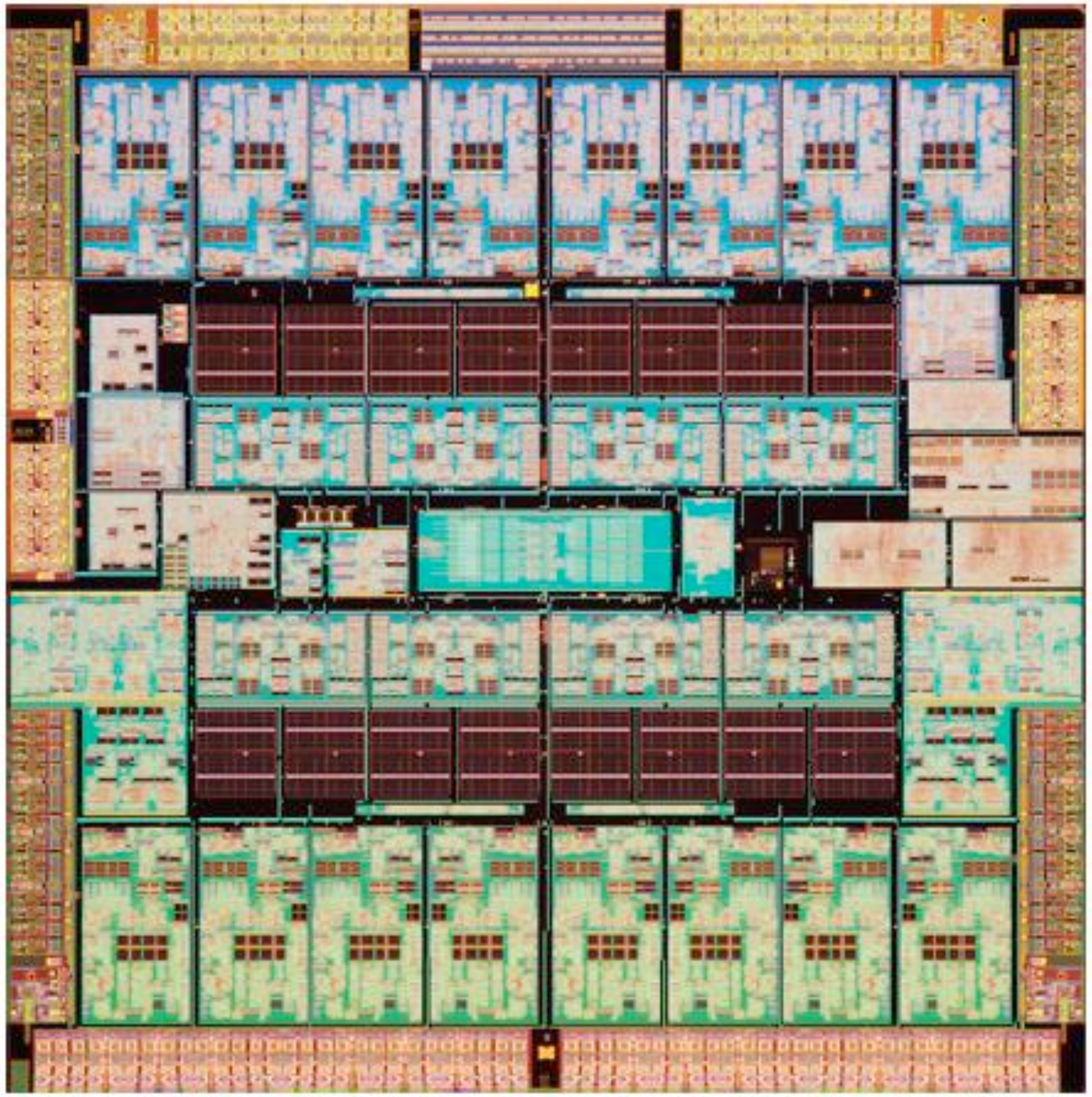


**Niagara 1** (2005)

8 x 4 = **32**

**Niagara 2** (2007)

8 x 8 = **64**



**Niagara 1** (2005)

8 x 4 = **32**

**Niagara 2** (2007)

8 x 8 = **64**

**Rainbow Falls**

16 x 8 = **128**

# Trends

# Trends

One core — Threads were for asynchrony, not parallelism

# Trends

One core — Threads were for asynchrony, not parallelism

Some cores — Coarse-grained parallelism usually enough
- Application-level requests were good task boundaries
- Thread pools were a reasonable scheduling mechanism

# Trends

One core — Threads were for asynchrony, not parallelism

Some cores — Coarse-grained parallelism usually enough

- Application-level requests were good task boundaries
- Thread pools were a reasonable scheduling mechanism

Many cores — Coarse-grained parallelism insufficient

- Application-level requests won't keep cores busy
- Shared work queues become a bottleneck

# Trends

One core — Threads were for asynchrony, not parallelism

Some cores — Coarse-grained parallelism usually enough
- Application-level requests were good task boundaries
- Thread pools were a reasonable scheduling mechanism

Many cores — Coarse-grained parallelism insufficient
- Application-level requests won't keep cores busy
- Shared work queues become a bottleneck

→ *Need to find finer-grained, CPU-intensive parallelism*

# The key challenges for multicore code

# The key challenges for multicore code

(1) Decompose problems into parallelizable work units

# The key challenges for multicore code

(1) Decompose problems into parallelizable work units

(2) Continue to meet (1) as the number of cores increases

# No silver bullet

# No silver bullet

Many point solutions:

# No silver bullet

Many point solutions:

- Work queues + thread pools

# No silver bullet

Many point solutions:

- Work queues + thread pools
- Divide & conquer (fork/join)

# No silver bullet

Many point solutions:

- Work queues + thread pools
- Divide & conquer (fork/join)
- Bulk data operations (select/map/reduce)

# No silver bullet

Many point solutions:

- Work queues + thread pools
- Divide & conquer (fork/join)
- Bulk data operations (select/map/reduce)
- Actors

# No silver bullet

Many point solutions:

- Work queues + thread pools
- Divide & conquer (fork/join)
- Bulk data operations (select/map/reduce)
- Actors
- Software transactional memory (STM)

# No silver bullet

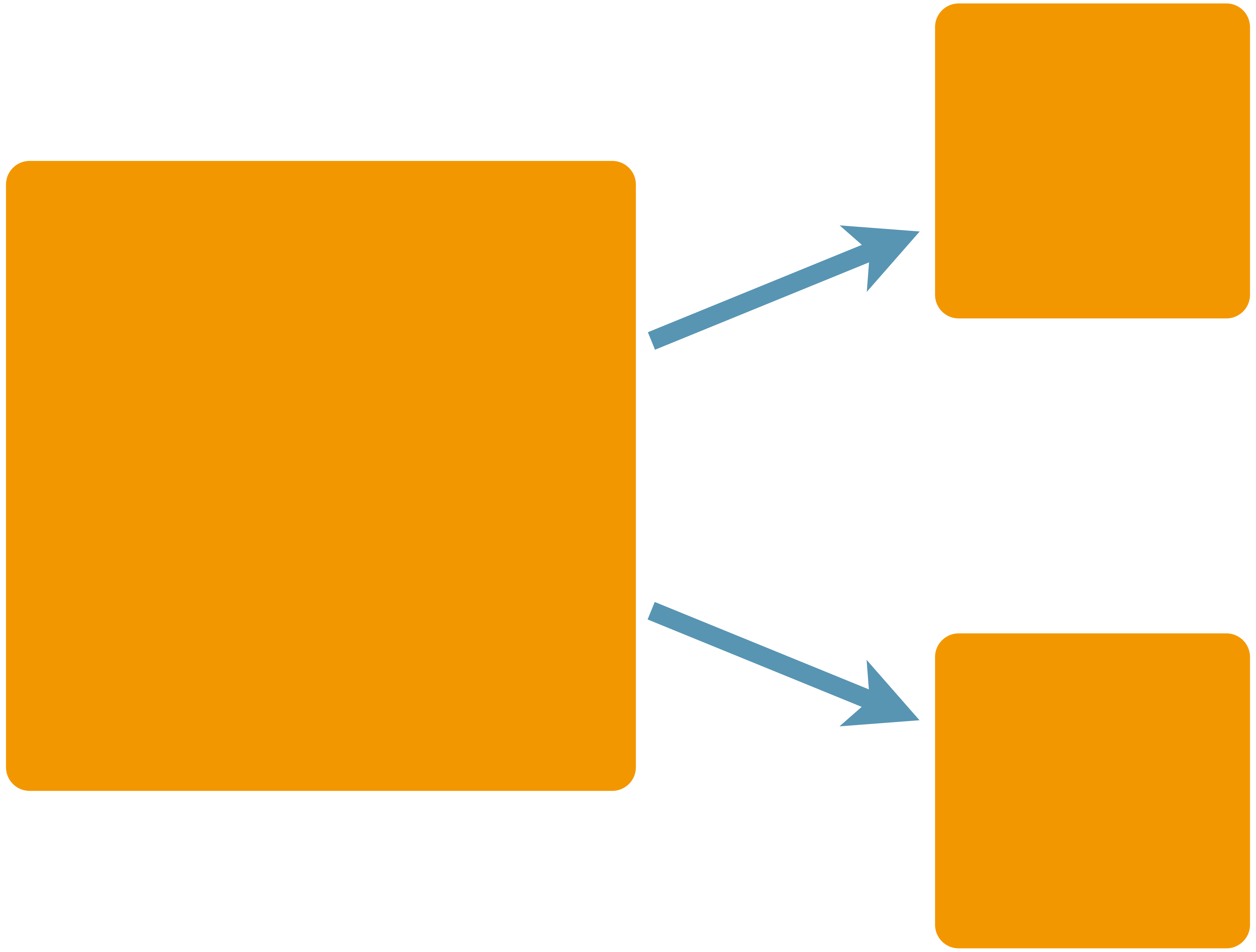
Many point solutions:

- Work queues + thread pools
- Divide & conquer (fork/join)
- Bulk data operations (select/map/reduce)
- Actors
- Software transactional memory (STM)
- GPU-based SIMD-style computation

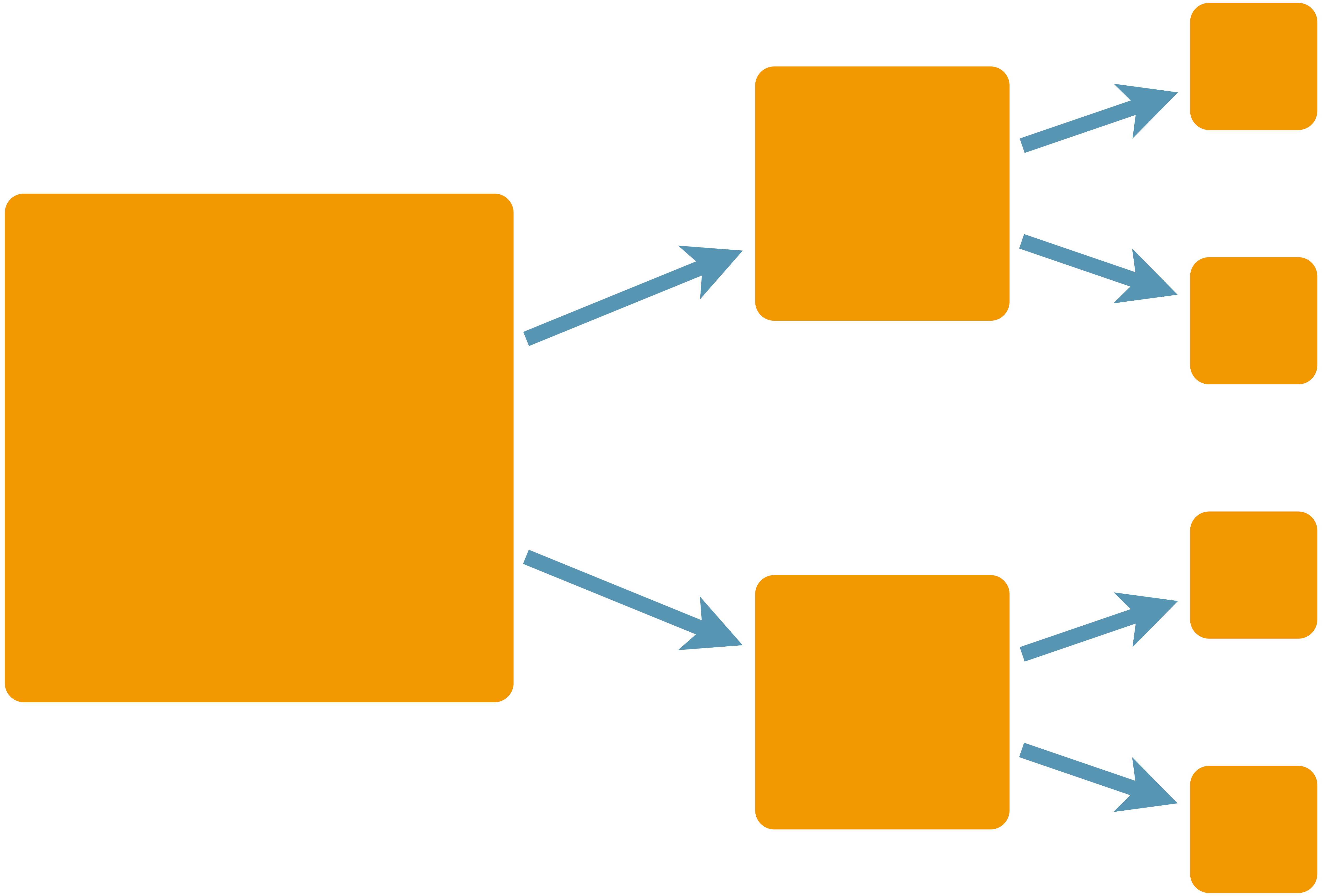
# Divide & conquer

# Divide & conquer

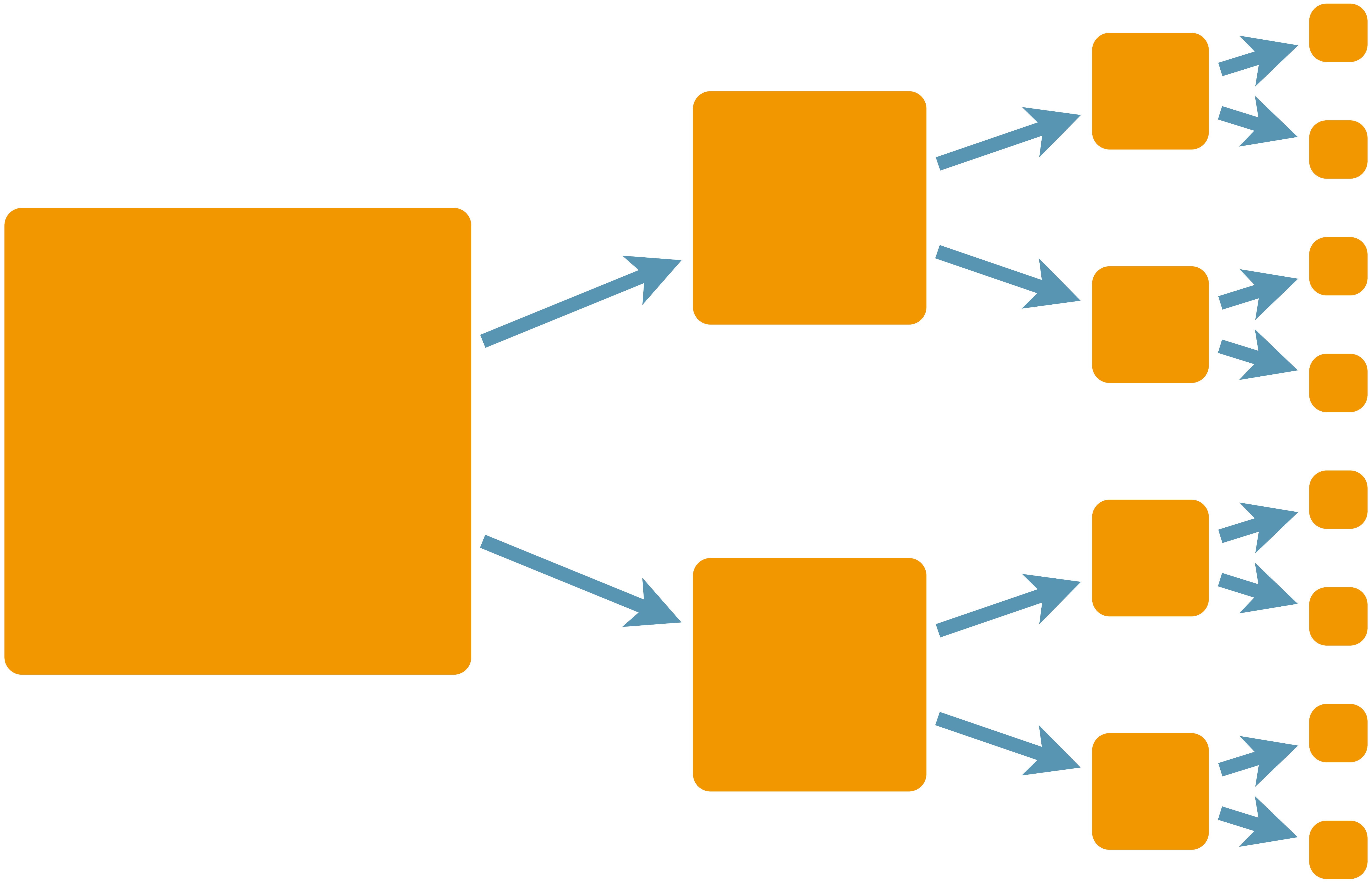
# Divide & conquer



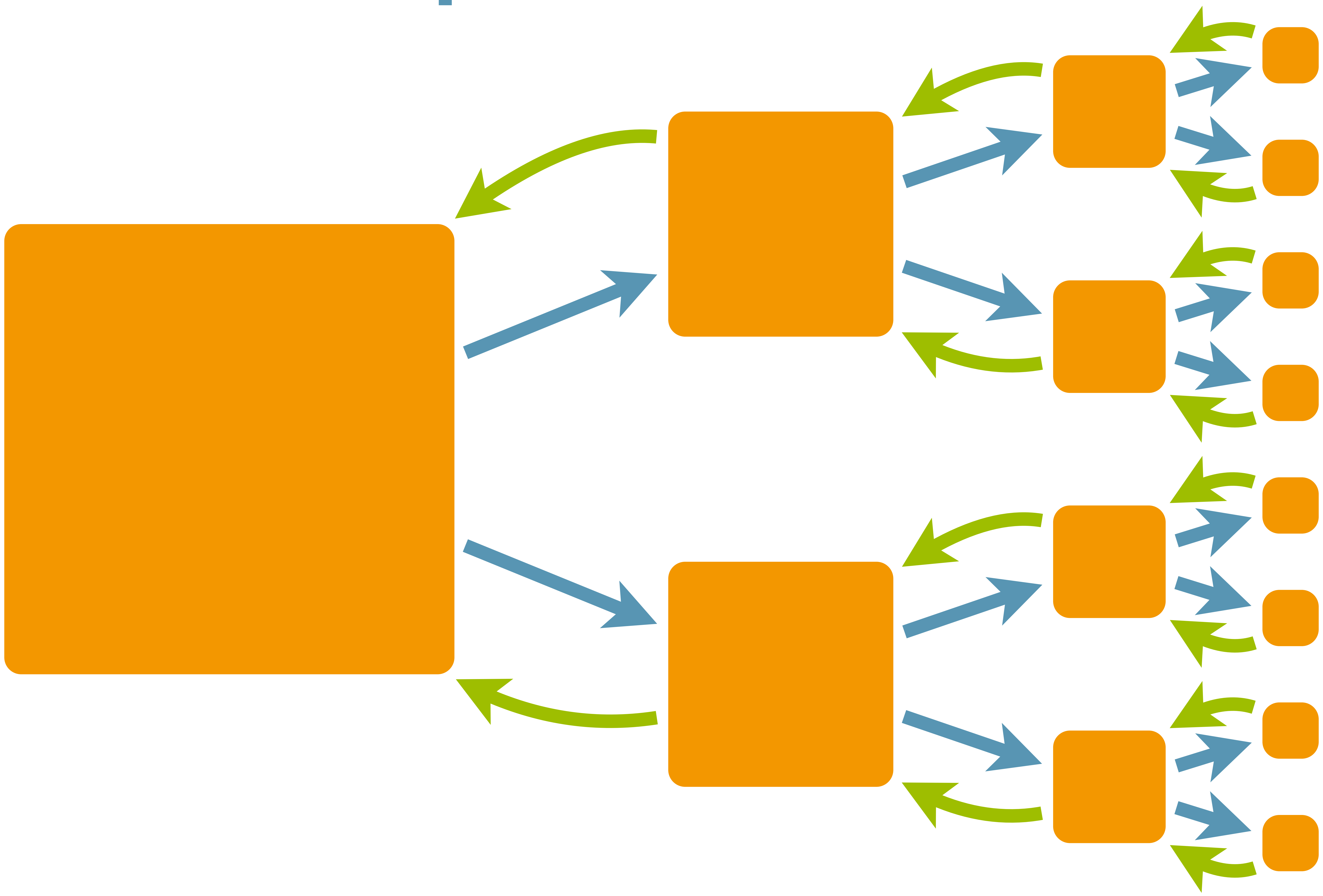
# Divide & conquer



# Divide & conquer



# Divide & conquer



```
Result solve(Problem p) {
    if (p.size() < SEQUENTIAL_THRESHOLD) {
        return p.solveSequentially();
    } else {
        int m = n / 2;
        Result left, right;
        INVOKE-IN-PARALLEL {
            left = solve(p.leftHalf());
            right = solve(p.rightHalf());
        }
        return combine(left, right);
    }
}
```

```
class Student {
    String name;
    int gradYear;
    double score;
}
```

```
class Student {
    String name;
    int gradYear;
    double score;
}

List<Student> students = ...;
```

```
class Student {
    String name;
    int gradYear;
    double score;
}

List<Student> students = ...;

double max = Double.MIN_VALUE;
for (Student s : students) {
    if (s.gradYear == 2010)
        max = Math.max(max, s.score);
}
```

```
class MaxFinder {

    final List<Student> students;

    MaxFinder(List<Student> ls) { students = ls; }

    double find() {
        double max = Double.MIN_VALUE;
        for (Student s : students) {
            if (s.gradYear == 2010)
                max = Math.max(max, s.score);
        }
        return max;
    }
```

ORACLE

```
class MaxFinder {

    final List<Student> students;

    MaxFinder(List<Student> ls) { students = ls; }

    double find() {
        double max = Double.MIN_VALUE;
        for (Student s : students) {
            if (s.gradYear == 2010)
                max = Math.max(max, s.score);
        }
        return max;
    }

    MaxFinder subFinder(int s, int e) {
        return new MaxFinder(students.subList(s, e));
    }
```

ORACLE

```
// Fork/join framework
import java.util.concurrent.*;
```

```
// Fork/join framework
import java.util.concurrent.*;

class MaxFinderTask
    extends RecursiveAction
{

    final MaxFinder maxf;
    double result;

    MaxFinderTask(MaxFinder mf) { maxf = mf; }
```

```
class MaxFinderTask
    extends RecursiveAction
{

    protected void compute() {
        int n = maxf.students.size();
        if (n < SEQUENTIAL_THRESHOLD) {
            result = maxf.find();
        } else {
            int m = n / 2;
            MaxFinderTask left
                = new MaxFinderTask(maxf.subFinder(0, m));
            MaxFinderTask right
                = new MaxFinderTask(maxf.subFinder(m, n));
            invokeAll(left, right);
            result = Math.max(left.result, right.result);
        }
    }
```

```
class MaxFinder {

    double find() {
        double max = Double.MIN_VALUE;
        for (Student s : students) {
            if (s.gradYear == 2010)
                max = Math.max(max, s.score);
        }
        return max;
    }

    MaxFinder subFinder(int s, int e) {
        return new MaxFinder(students.subList(s, e));
    }
```

```java
class MaxFinder {

    double find() { ... }

    MaxFinder subFinder(int s, int e) {
        return new MaxFinder(students.subList(s, e));
    }

    double parallelFind() {
        MaxFinderTask mft = new MaxFinderTask(this);
        ForkJoinPool pool = new ForkJoinPool();
        pool.invoke(mft);
        return mft.result;
    }
```

ORACLE

```
class MaxFinderTask
    extends RecursiveAction
{

    protected void compute() {
        int n = maxf.students.size();
        if (n < SEQUENTIAL_THRESHOLD) {
            result = maxf.find();
        } else {
            int m = n / 2;
            MaxFinderTask left
                = new MaxFinderTask(maxf.subFinder(0, m));
            MaxFinderTask right
                = new MaxFinderTask(maxf.subFinder(m, n));
            invokeAll(left, right);
            result = Math.max(left.result, right.result);
        }
    }
```

```
class MaxFinderTask
    extends RecursiveAction
{

    protected void compute() {
        int n = maxf.students.size();
        if (n < SEQUENTIAL_THRESHOLD) {     // ???
            result = maxf.find();
        } else {
            int m = n / 2;
            MaxFinderTask left
                = new MaxFinderTask(maxf.subFinder(0, m));
            MaxFinderTask right
                = new MaxFinderTask(maxf.subFinder(m, n));
            invokeAll(left, right);
            result = Math.max(left.result, right.result);
        }
    }
```

# Performance considerations

# Performance considerations

- Choosing the sequential threshold
  - Smaller tasks increase parallelism
  - Larger tasks reduce coordination overhead
  - Ultimately you must profile your code

# Performance considerations

- Choosing the sequential threshold
  - Smaller tasks increase parallelism
  - Larger tasks reduce coordination overhead
  - Ultimately you must profile your code

| *Sequential threshold* | 500K | 50K | 5K | 500 | 50 |
|---|---|---|---|---|---|
| Dual Xeon HT (4) | 0.88 | 3.02 | 3.20 | 2.22 | 0.43 |
| 8-way Opteron (8) | 1.00 | 5.29 | 5.73 | 4.53 | 2.03 |
| 8-core Niagara (32) | 0.98 | 10.46 | 17.21 | 15.34 | 6.49 |

# Performance considerations

# Performance considerations

- The fork/join framework minimizes per-task overhead for *compute-intensive* tasks
  - Not recommended for tasks that mix CPU and I/O activity

# Performance considerations

- The fork/join framework minimizes per-task overhead for *compute-intensive* tasks
  - Not recommended for tasks that mix CPU and I/O activity
- A portable way to express many parallel algorithms
  - Code is independent of execution topology
  - Reasonably efficient for a wide range of core counts
  - Library-managed parallelism

# No silver bullet—*but many useful tools*

Many point solutions:

- Work queues + thread pools
- Divide & conquer (fork/join)
- Bulk data operations (select/map/reduce)
- Actors
- Software transactional memory (STM)
- GPU-based SIMD-style computation

Java™