15-210: Parallelism in the Real World

- Types of paralellism
- Parallel Thinking
- Nested Parallelism
- Examples (Cilk, OpenMP, Java Fork/Join)
- Concurrency

Cray-1 (1976): the world's most expensive love seat

Data Center: Hundred's of thousands of computers

Since 2005: Multicore computers

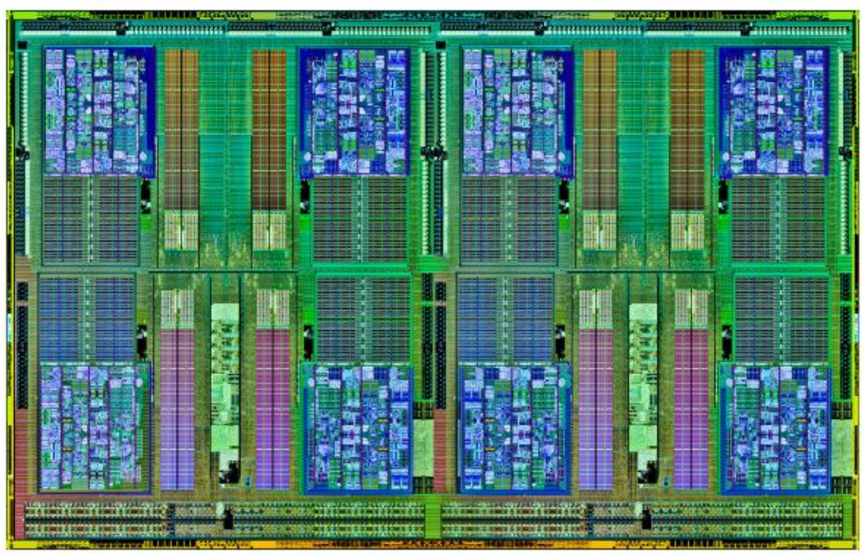
AMD Opteron (sixteen-core) Model 6274
by AMD
(1 customer review)

List Price: ~~$693.00~~
Price: $599.99 Prime
You Save: $93.01 (13%)

Only 1 left in stock (more on the way).
Ships from and sold by **Amazon.com**. Gift-wrap available.

Want it delivered Monday, November 5? Order it in the next 14 hours and 37 minutes
Delivery may be impacted by Hurricane Sandy. Proceed to checkout to see estimated

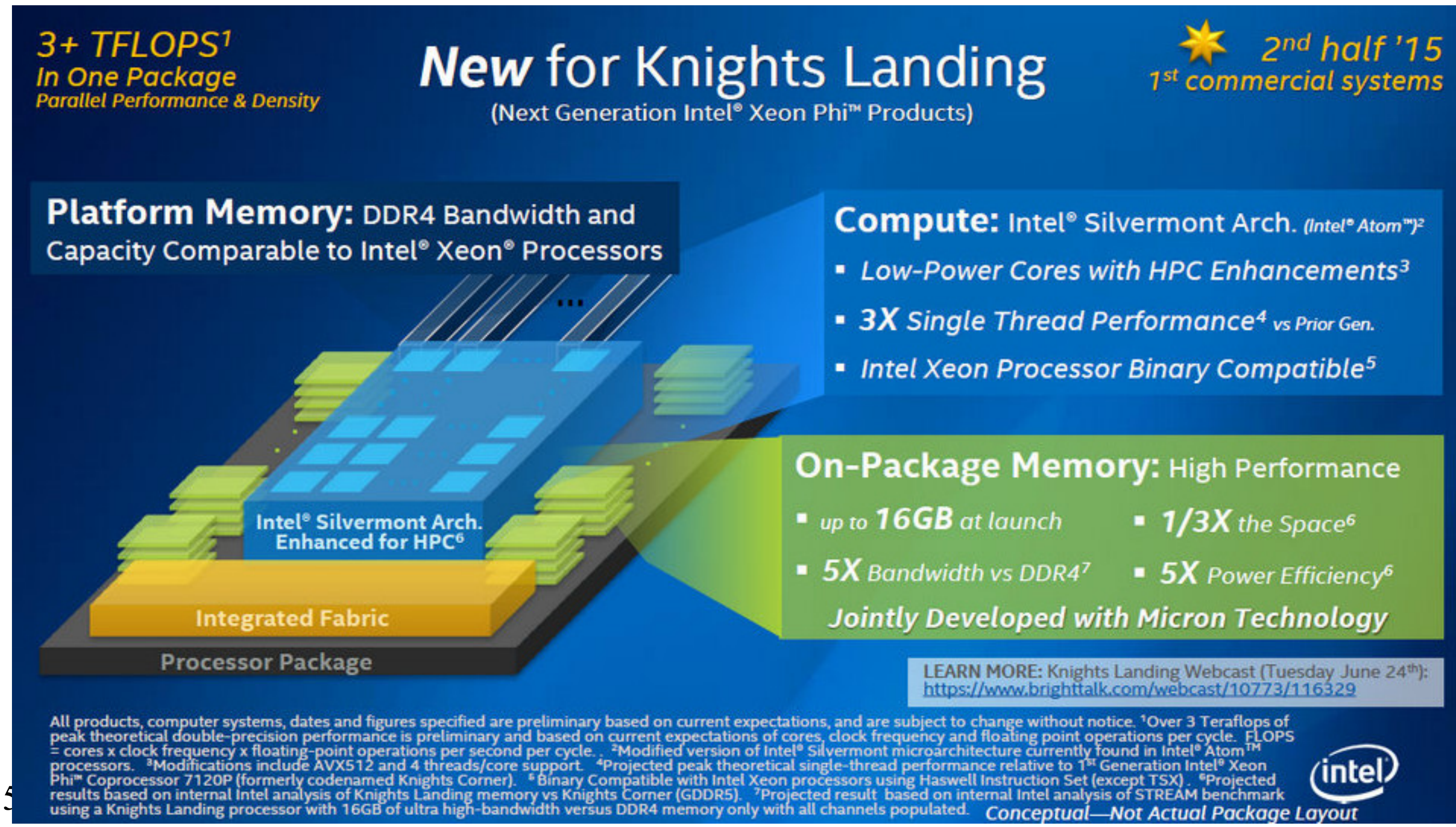
43 new from $599.99



Xeon Phi: Knights Landing

72 Cores, x86, 500Gbytes of memory bandwidth

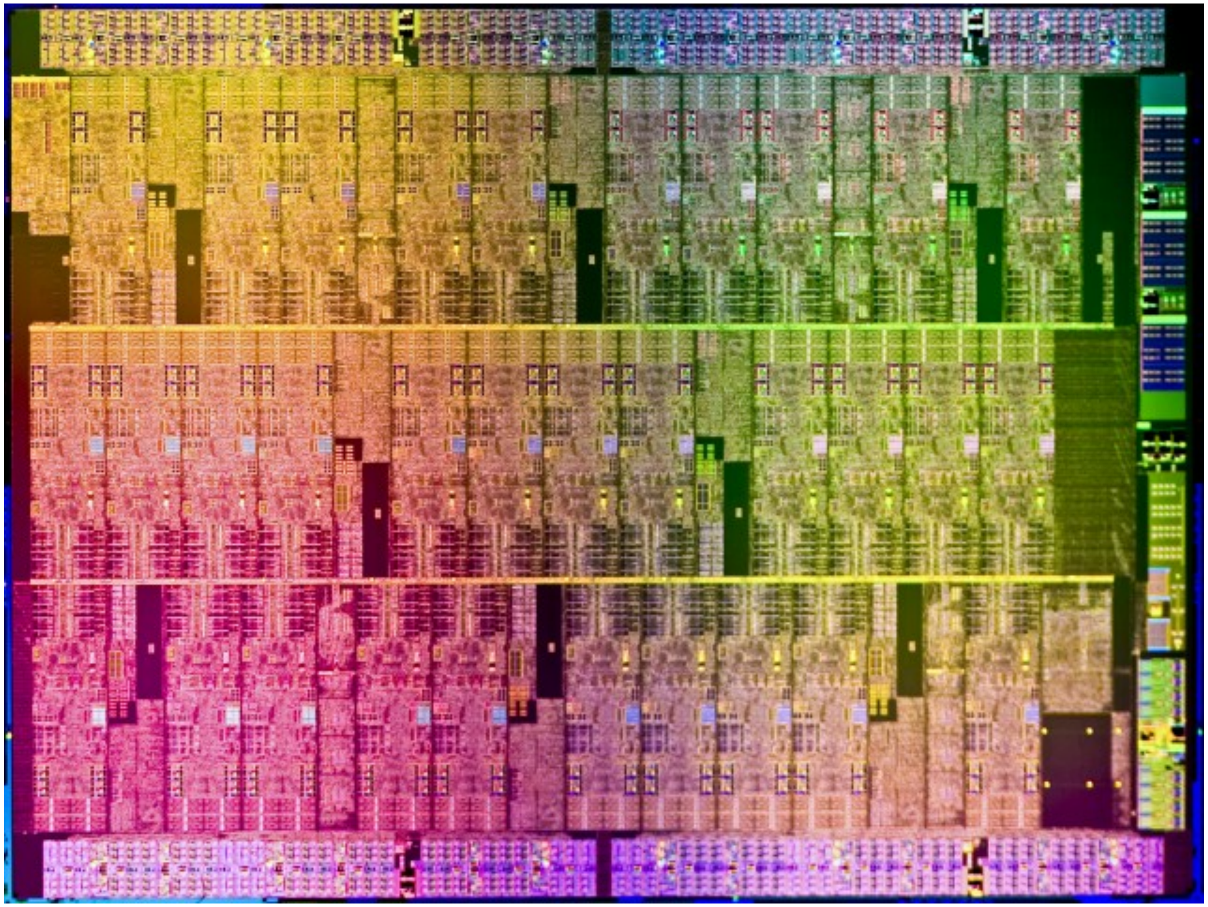
Summer 2015



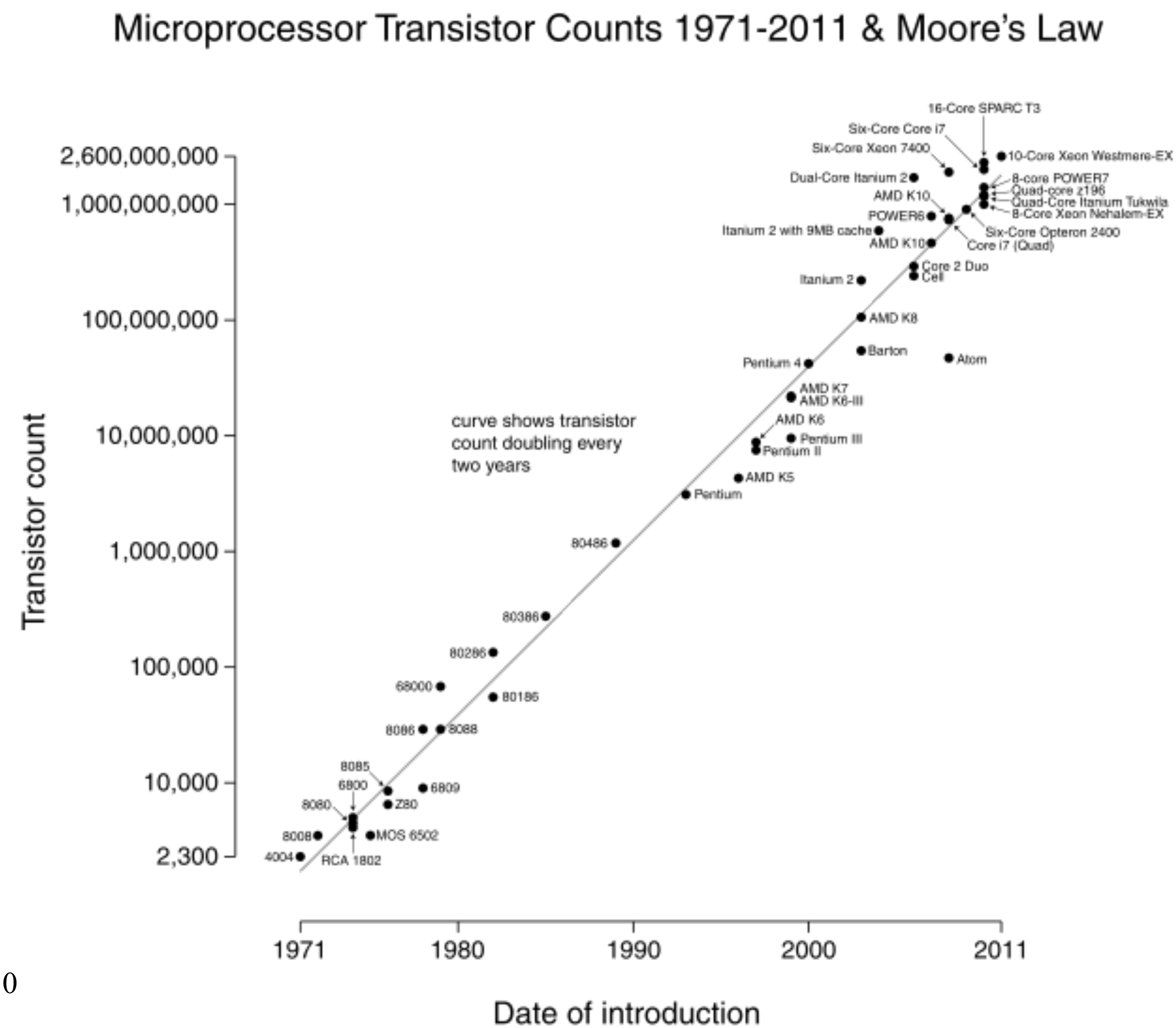
Xeon Phi: Knights Landing

72 Cores, x86, 500Gbytes of memory bandwidth

Summer 2015

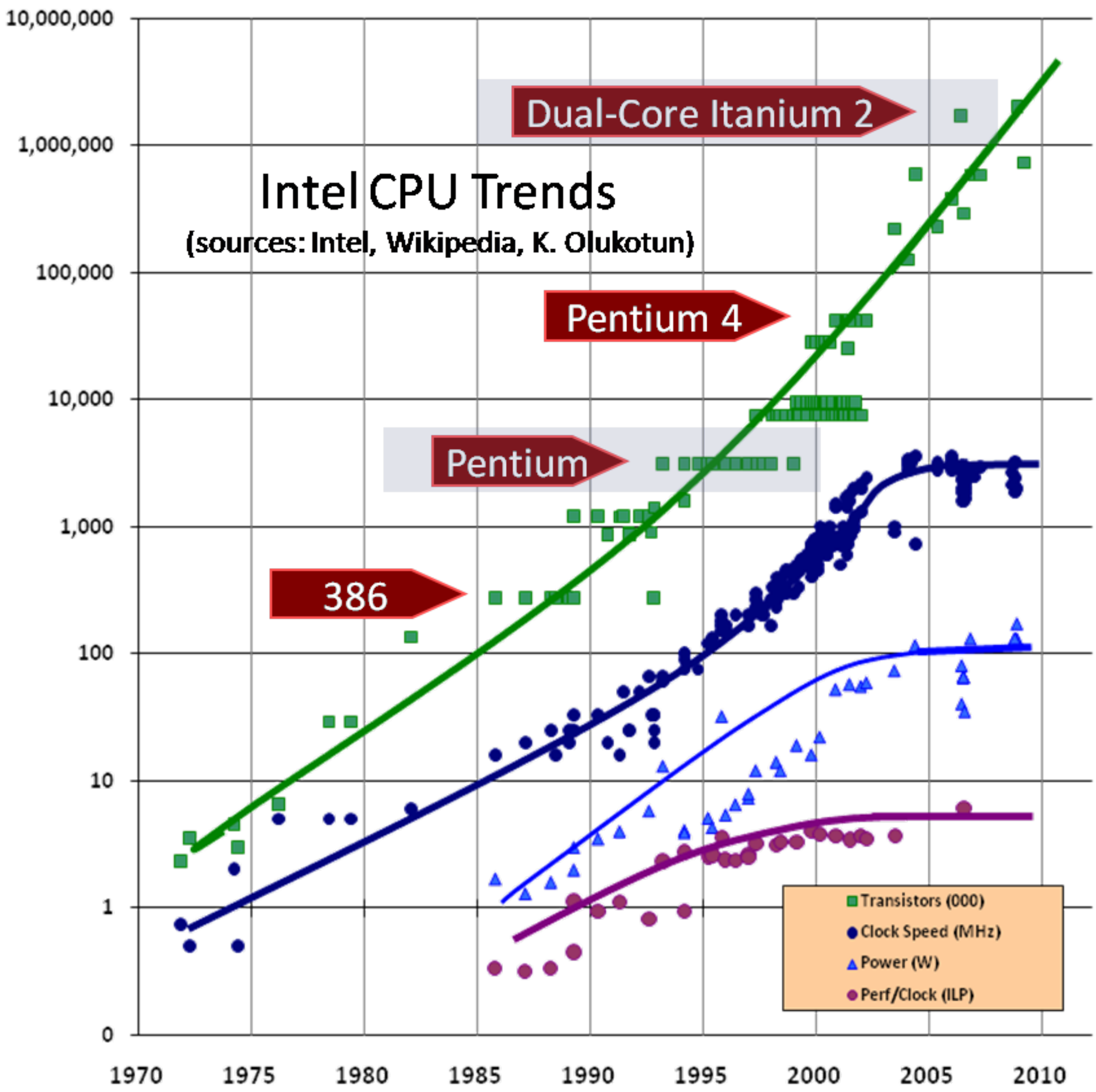


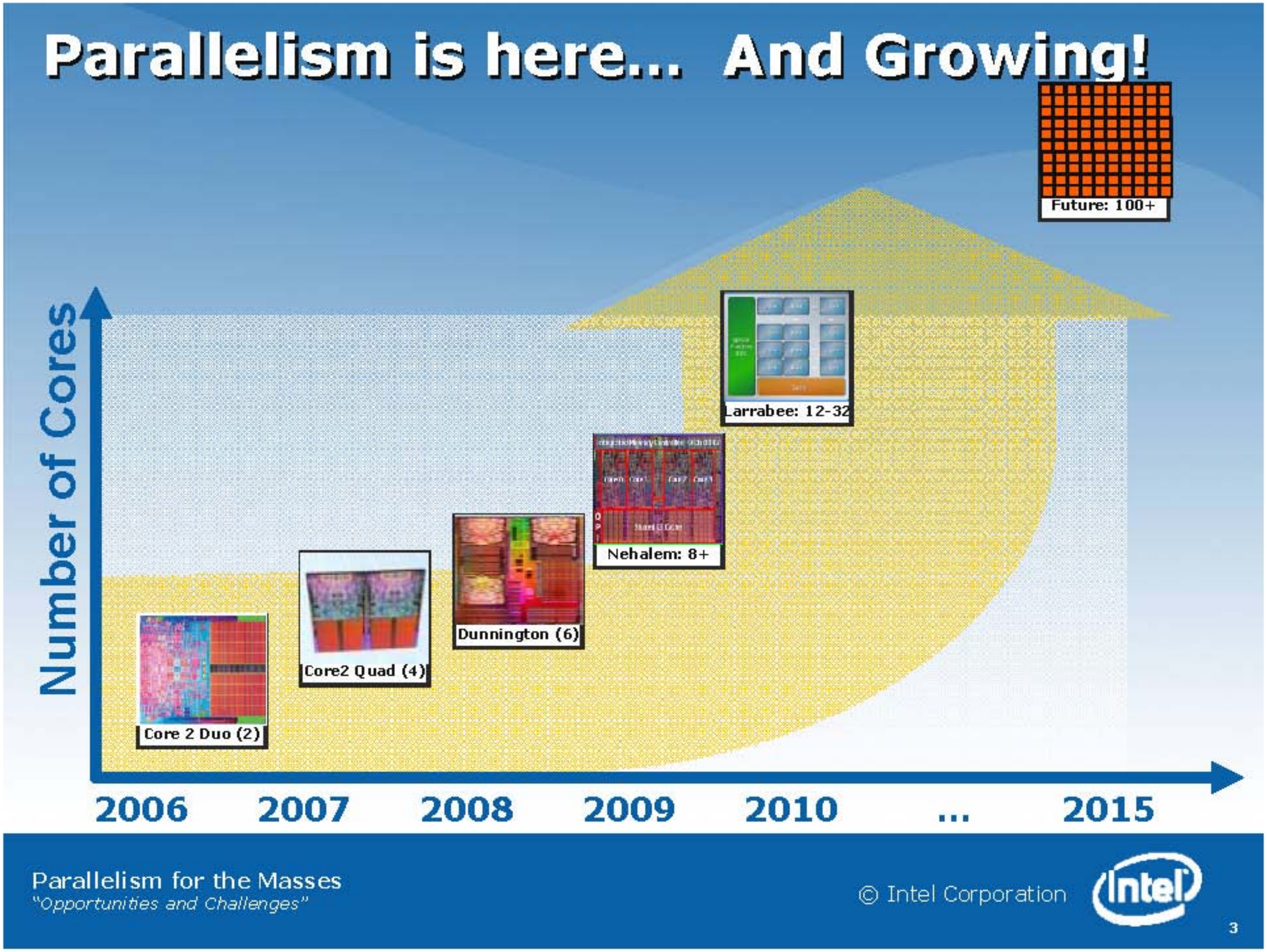
Moore's Law



Microprocessor Transistor Counts 1971-2011 & Moore's Law
Transistor count
Date of introduction
2,600,000,000
1,000,000,000
100,000,000
10,000,000
1,000,000
100,000
10,000
2,300
1971
1980
1990
2000
2011
curve shows transistor count doubling every two years
16-Core SPARC T3
Six-Core Core i7
Six-Core Xeon 7400
10-Core Xeon Westmere-EX
Dual-Core Itanium 2
8-core POWER7
Quad-core z196
Quad-Core Itanium Tukwila
8-Core Xeon Nehalem-EX
AMD K10
POWER6
Itanium 2 with 9MB cache
Six-Core Opteron 2400
AMD K10
Core i7 (Quad)
Core 2 Duo
Cell
Itanium 2
AMD K8
Barton
Atom
Pentium 4
AMD K7
AMD K6-III
AMD K6
Pentium III
Pentium II
AMD K5
Pentium
80486
80386
80286
68000
80186
8086
8088
8085
6800
6809
8080
Z80
8008
MOS 6502
4004
RCA 1802

Moore's Law and Performance





 Andrew Chien, 2008

64 core blade servers ($6K) (shared memory)

AMD Opteron (sixteen-core) Model 6274

by AMD

(1 customer review)

List Price: ~~$693.00~~

Price: $599.99 ✓Prime

You Save: $93.01 (13%)

Only 1 left in stock (more on the way).

Ships from and sold by **Amazon.com**. Gift-wrap available.

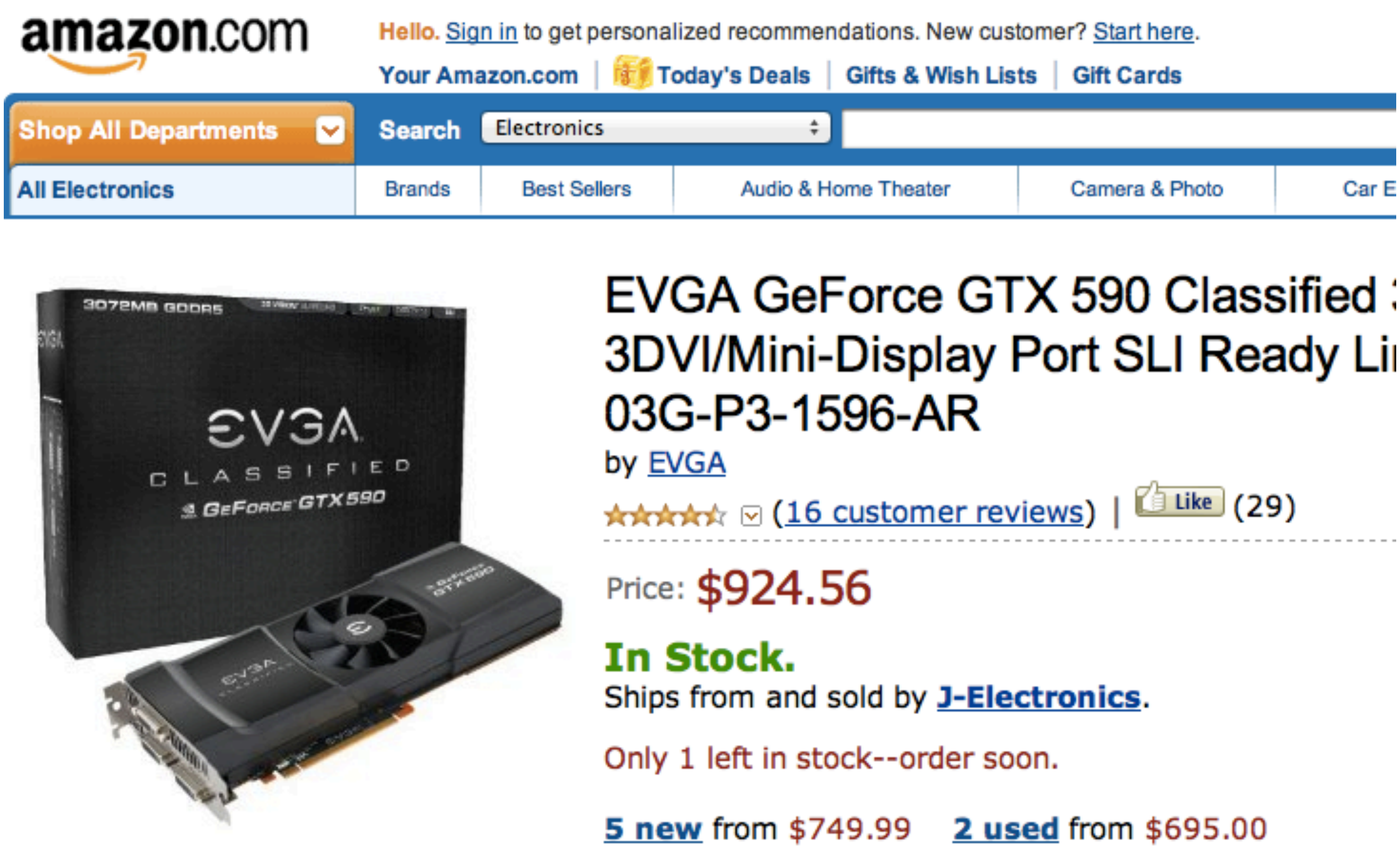
Want it delivered Monday, November 5? Order it in the next 14 hours and 37 minutes

Delivery may be impacted by Hurricane Sandy. Proceed to checkout to see estimated

43 new from $599.99

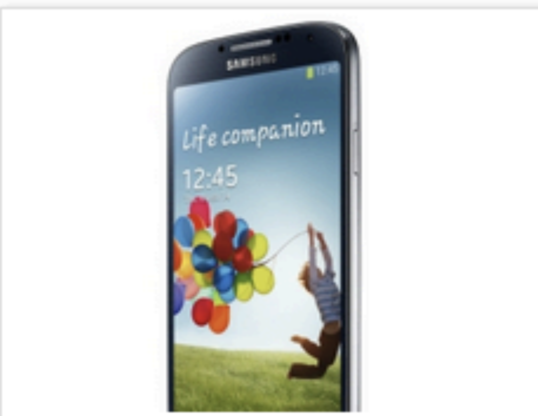
x 4 =

1024 "cuda" cores



Samsung Galaxy S IV is now Official: Octa-Core CPU, 5″ Full HD Display & 13MP Camera

Follow: Phones GT-I9500 Samsung Display Samsung Exynos Samsung Galaxy S IV Samsung Mobile Unpacked 2013



Samsung has just announced the Samsung Galaxy S4 at their Mobile Unpacked Event 2013 Episode 1 in New York, USA. The Galaxy S4 features a stunning 4.99″ Full HD (1920×1080) SuperAMOED display. With a 441 ppi pixel density, your eyes won't be able to distinguish the pixels, which ensures excellent visual comfort. Even though the Galaxy S4 has a large display and a massive battery of 2,600 MAh, it's only 7.9mm thick. Samsung's latest flagship device is PACKED with powerful components, consisting of Samsung's latest Exynos 5 Octa-Core (5410) CPU based on ARM's big.LITTLE technology with Quad Cortex-

Intel Has a 48-Core Chip for Smartphones and Tablets

By Wolfgang Gruener OCTOBER 31, 2012 9:20 AM - Source: Computerworld

Intel has developed a prototype of a 48-core processor for smartphones. Before you ask: No, you can't buy a 48-core smartphone next year.

Parallel Hardware

Many forms of parallelism

- Supercomputers: large scale, shared memory
- Clusters and data centers: large-scale, distributed memory
- Multicores: tightly coupled, smaller scale
- GPUs, on chip vector units
- Instruction-level parallelism

Parallelism is important in the real world.

Key Challenge: Software (How to Write Parallel Code?)

At a high-level, it is a two step process:

- Design a work-efficient, low-span parallel algorithm
- Implement it on the target hardware

In reality: each system required different code because programming systems are immature

- Huge effort to generate efficient parallel code.
  - Example: Quicksort in MPI is 1700 lines of code, and about the same in CUDA
- Implement one parallel algorithm: a whole thesis.

Take 15-418 (Parallel Computer Architecture and Prog.)

15-210 Approach

Enable parallel thinking by raising abstraction level

I. Parallel thinking: Applicable to many machine models and programming languages

II. Reason about correctness and efficiency of algorithms and data structures.

Parallel Thinking

Recognizing true dependences: unteach sequential programming.

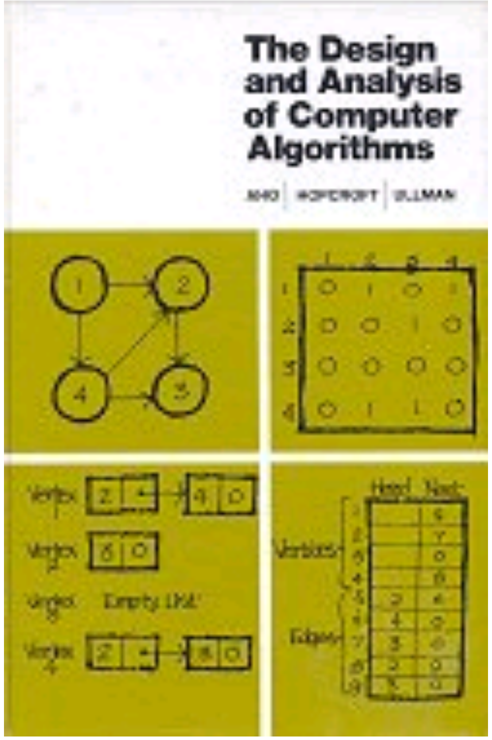
Parallel algorithm-design techniques

- Operations on aggregates: map/reduce/scan
- Divide & conquer, contraction
- Viewing computation as DAG (based on dependences)

Cost model based on work and span

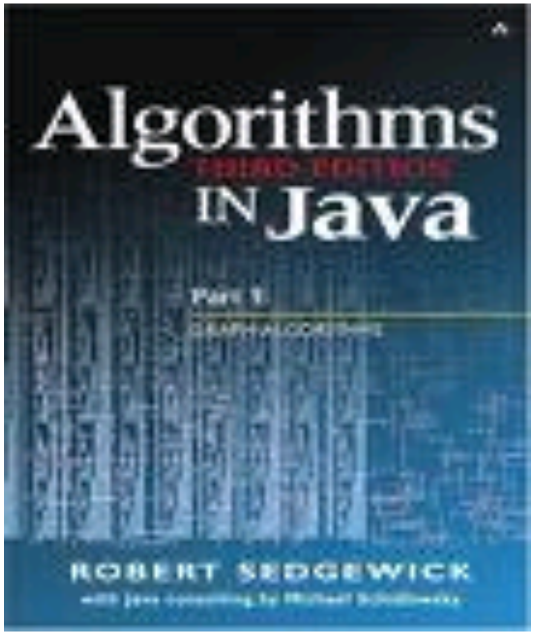
Quicksort from Aho-Hopcroft-Ullman (1974)

procedure QUICKSORT(**S**):
  if S contains at most one element **then return S**
  else
    begin
    choose an element **a** randomly from **S**;
    let S_1, S_2 and S_3 be the sequences of
      elements in **S** less than, equal to,
      and greater than **a**, respectively;
    return (QUICKSORT(S_1) followed by S_2
      followed by QUICKSORT(S_3))
end



Quicksort from Sedgewick (2003)

```
public void quickSort(int[] a, int left, int right) {
    int i = left-1;  int j = right;
    if (right <= left) return;
    while (true) {
      while (a[++i] < a[right]);
      while (a[right] < a[--j])
        if (j==left) break;
      if (i >= j) break;
      swap(a,i,j); }
    swap(a, i, right);
    quickSort(a, left, i - 1);
    quickSort(a, i+1, right); }
```



Styles of Parallel Programming

Data parallelism/Bulk Synchronous/SPMD

Nested parallelism : what we covered

Message passing

Futures (other pipelined parallelism)

General Concurrency

Nested Parallelism

Nested Parallelism =

arbitrary nesting of parallel loops + fork-join

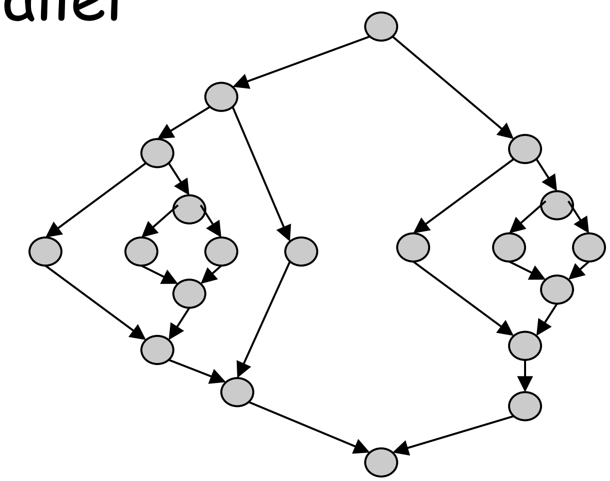
- Assumes no synchronization among parallel tasks except at joint points.
- Deterministic if no race conditions

Advantages:

- Good schedulers are known
- Easy to understand, debug, and analyze costs
- Purely functional, or imperative…either works

Serial Parallel DAGs

Dependence graphs of nested parallel computations are series parallel



Two tasks are parallel if not reachable from each other.

A data race occurs if two parallel tasks are involved in a race if they access the same location and at least one is a write.

Nested Parallelism: parallel loops

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

Cilk

```
Parallel.ForEach(A, x => x+1);
```

Microsoft TPL (C#,F#)

```
B = {x + 1 : x in A}
```

Nesl, Parallel Haskell

```
#pragma omp for
for (i=0; i < n; i++)
  B[i] = A[i] + 1;
```

OpenMP

Nested Parallelism: fork-join

```
cobegin {
  S1;
  S2;}
```

Dates back to the 60s. Used in dialects of Algol, Pascal

```
coinvoke(f1,f2)
Parallel.invoke(f1,f2)
```

Java fork-join framework
Microsoft TPL (C#,F#)

```
#pragma omp sections
{
  #pragma omp section
  S1;
  #pragma omp section
  S2;
}
```

OpenMP (C++, C, Fortran, …)

Nested Parallelism: fork-join

```
spawn S1;
S2;
sync;
```

cilk, cilk+

```
(exp1 || exp2)
```

Various functional languages

```
plet
  x = exp1
  y = exp2
in
  exp3
```

Various dialects of ML and Lisp

Cilk vs. what we've covered

ML: val (a,b) = par(fn () => f(x), fn () => g(y))

Psuedocode: val (a,b) = (f(x) || g(y))

Cilk:

```
cilk_spawn a = f(x);
b = g(y);
cilk_sync;
```

Fork Join

ML: S = map f A

Psuedocode: S = <f x : x in A>

Cilk:

```
cilk_for (int i = 0; i < n; i++)
   S[i] = f(A[i])
```

Map

Cilk vs. what we've covered

ML: S = tabulate f n

Psuedocode: S = <f i : i in <0,..n-1>>

Tabulate

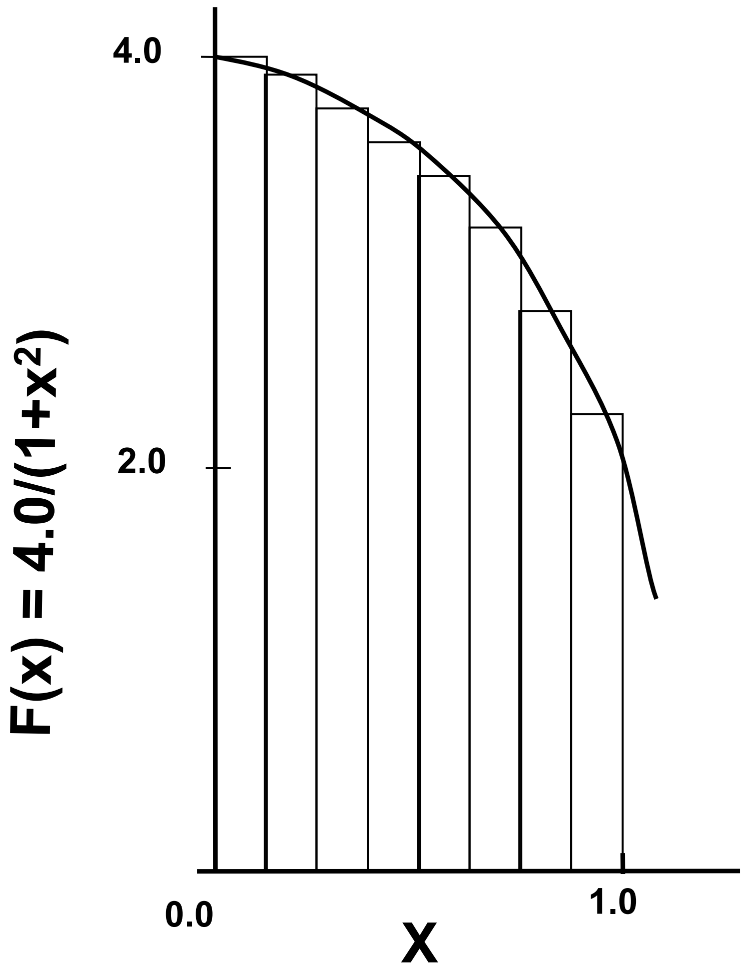
Cilk:

```
cilk_for (int i = 0; i < n; i++)
    S[i] = f(i)
```

Example Cilk

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

Example OpenMP: Numerical Integration



Mathematically, we know that:

$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx = \pi$$

We can approximate the integral as a sum of rectangles:

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

where each rectangle has width Δx and height $F(x_i)$ at the middle of interval i.

The C code for Approximating PI

```
static long num_steps = 100000;
double step;
void main ()
{           int i;   double x, pi, sum = 0.0;

            step = 1.0/(double) num_steps;
             x = 0.5 * step;
            for (i=0;i<= num_steps; i++){
                      x+=step;
                      sum += 4.0/(1.0+x*x);
            }
            pi = step * sum;
}
```

The C/openMP code for Approx. PI

```
#include <omp.h>
static long num_steps = 100000;        double step;
void main ()
{     int i;              double x, pi, sum = 0.0;
      step = 1.0/(double) num_steps;
#pragma omp   parallel    for    private(i, x) reduction(+:sum)
      for (i=0;i<= num_steps; i++){
              x = (i+0.5)*step;
              sum = sum + 4.0/(1.0+x*x);
      }
      pi = step * sum;
}
```

Private clause creates data local to a thread

Reduction used to manage dependencies

Example : Java Fork/Join

```
class Fib extends FJTask {
   volatile int result; // serves as arg and result
   int n;
   Fib(int _n) { n = _n; }

   public void run() {
      if (n <= 1) result = n;
      else if (n <= sequentialThreshold) number = seqFib(n);
      else {
         Fib f1 = new Fib(n - 1);
         Fib f2 = new Fib(n - 2);
         coInvoke(f1, f2);
         result = f1.result + f2.result;
      }
   }
}
```

Cost Model (General)

Compositional:

Work : total number of operations

- costs are added across parallel calls

Span : depth/critical path of the computation

- Maximum span is taken across forked calls

Parallelism = Work/Span

- Approximately # of processors that can be effectively used.

Combining costs (Nested Parallelism)

Combining for parallel for:

```
pfor (i=0; i<n; i++)
  f(i);
```

$$W_{\text{pexp}}(\text{pfor} \ldots) = \sum_{i=0}^{n-1} W_{\text{exp}}(\text{f(i)}) \qquad \textbf{work}$$

$$D_{\text{pexp}}(\text{pfor} \ldots) = \max_{i=0}^{n-1} D_{\text{exp}}(\text{f(i)}) \qquad \textbf{span}$$

Why Work and Span

Simple measures that give us a good sense of efficiency (work) and scalability (span).

Can schedule in $O(W/P + D)$ time on P processors.

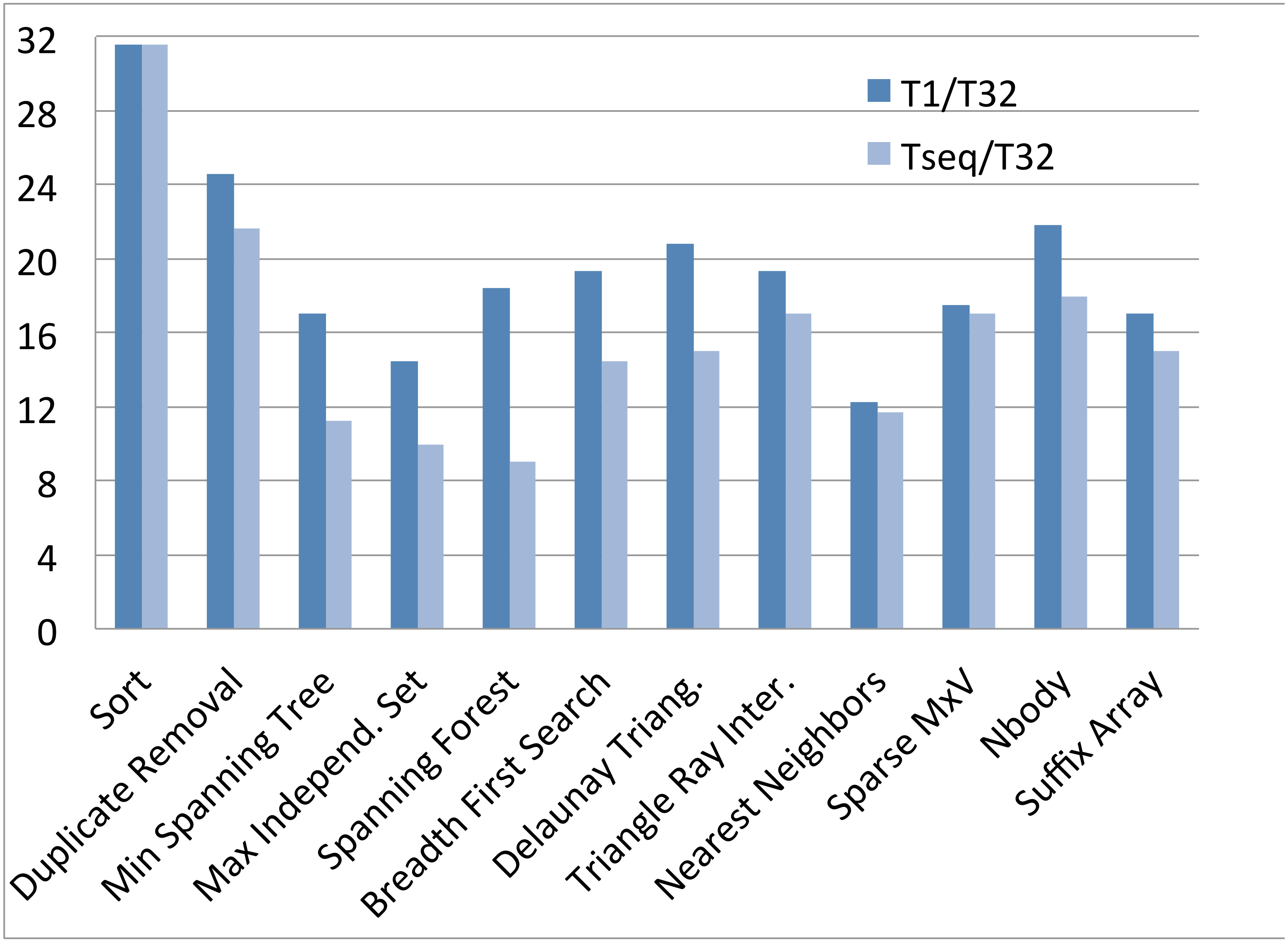
This is within a constant factor of optimal.

Goals in designing an algorithm

1. Work should be about the same as the sequential running time. When it matches asymptotically we say it is **work efficient.**
2. Parallelism (W/D) should be polynomial. $O(n^{1/2})$ is probably good enough

How do the problems do on a modern multicore

2
1.6
1.6
1.2
10
9
4.5
15
17
1.7
17
18
15



Styles of Parallel Programming

- Data parallelism/Bulk Synchronous/SPMD
- Nested parallelism : what we covered
- Message passing
- Futures (other pipelined parallelism)
- General Concurrency

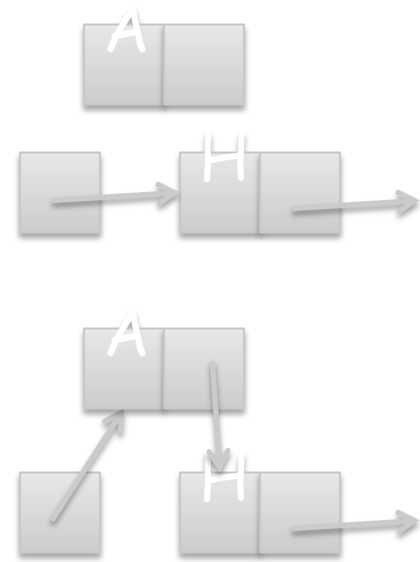
Parallelism vs. Concurrency

- Parallelism: using multiple processors/cores running at the same time. Property of the machine
- Concurrency: non-determinacy due to interleaving threads. Property of the application.

		Concurrency	
		sequential	concurrent
Parallelism	serial	Traditional programming	Traditional OS
	parallel	Deterministic parallelism	General parallelism

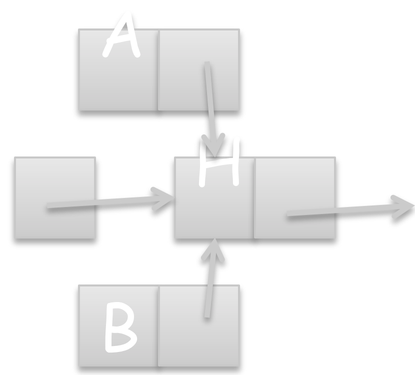
Concurrency : Stack Example 1

```
struct link {int v; link* next;}

struct stack {
  link* headPtr;

  void push(link* a) {
    a->next = headPtr;
    headPtr = a;    }

  link* pop() {
    link* h = headPtr;
    if (headPtr != NULL)
      headPtr = headPtr->next;
    return h;}
}
```



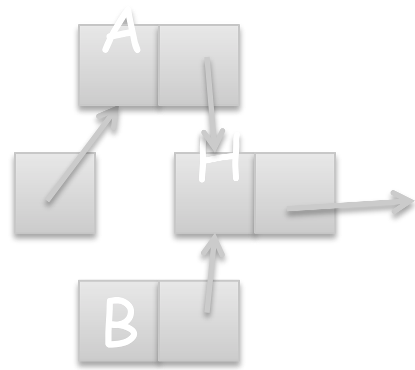
Concurrency : Stack Example 1

```
struct link {int v; link* next;}

struct stack {
  link* headPtr;

  void push(link* a) {
    a->next = headPtr;
    headPtr = a;    }

  link* pop() {
    link* h = headPtr;
    if (headPtr != NULL)
      headPtr = headPtr->next;
    return h;}
}
```



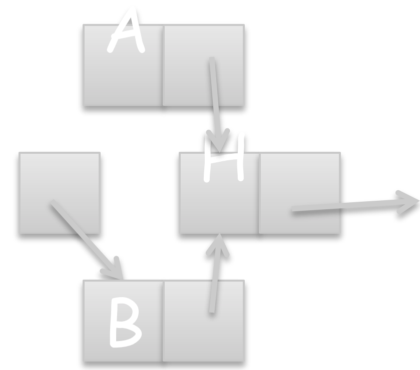
Concurrency : Stack Example 1

```
struct link {int v; link* next;}

struct stack {
  link* headPtr;

  void push(link* a) {
    a->next = headPtr;
    headPtr = a;    }

  link* pop() {
    link* h = headPtr;
    if (headPtr != NULL)
      headPtr = headPtr->next;
    return h;}
}
```



Concurrency : Stack Example 1

```
struct link {int v; link* next;}

struct stack {
  link* headPtr;

  void push(link* a) {
    a->next = headPtr;
    headPtr = a;    }

  link* pop() {
    link* h = headPtr;
    if (headPtr != NULL)
      headPtr = headPtr->next;
    return h;}
}
```



CAS

```
bool CAS(ptr* addr, ptr a, ptr b) {
  atomic {
    if (*addr == a) {
      *addr = b;
      return 1;
    } else
      return 0;
  }
}
```

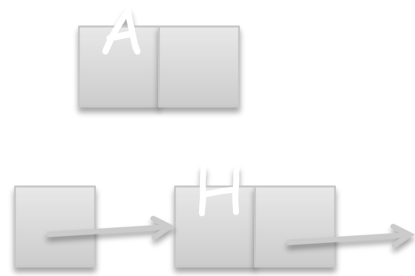
A built in instruction on most processors:

CMPXCHG8B – 8 byte version for x86

CMPXCHG16B – 16 byte version

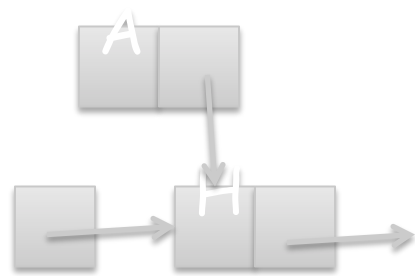
Concurrency : Stack Example 2

```
struct stack {
  link* headPtr;

  void push(link* a) {
    do {
      link* h = headPtr;
      a->next = h;
    while (!CAS(&headPtr, h, a)); }

  link* pop() {
    do {
      link* h = headPtr;
      if (h == NULL) return NULL;
      link* nxt = h->next;
    while (!CAS(&headPtr, h, nxt))}
    return h;}
}
```



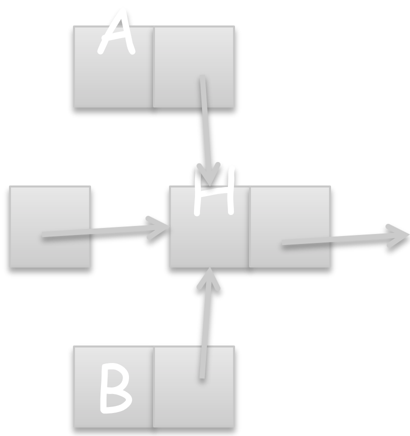
Concurrency : Stack Example 2

```
struct stack {
  link* headPtr;
  void push(link* a) {
    do {
      link* h = headPtr;
      a->next = h;
    while (!CAS(&headPtr, h, a)); }
  link* pop() {
    do {
      link* h = headPtr;
      if (h == NULL) return NULL;
      link* nxt = h->next;
    while (!CAS(&headPtr, h, nxt))}
    return h;}
}
```



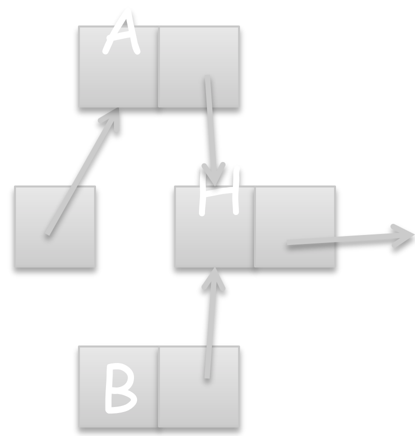
Concurrency : Stack Example 2

```
struct stack {
  link* headPtr;

  void push(link* a) {
    do {
      link* h = headPtr;
      a->next = h;
    while (!CAS(&headPtr, h, a)); }

  link* pop() {
    do {
      link* h = headPtr;
      if (h == NULL) return NULL;
      link* nxt = h->next;
    while (!CAS(&headPtr, h, nxt))}
    return h;}
}
```



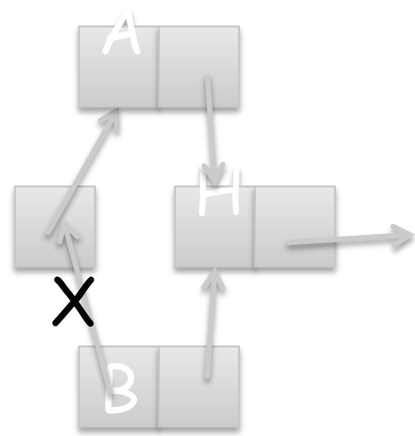
Concurrency : Stack Example 2

```
struct stack {
  link* headPtr;

  void push(link* a) {
    do {
      link* h = headPtr;
      a->next = h;
    while (!CAS(&headPtr, h, a)); }

  link* pop() {
    do {
      link* h = headPtr;
      if (h == NULL) return NULL;
      link* nxt = h->next;
    while (!CAS(&headPtr, h, nxt))}
    return h;}
}
```



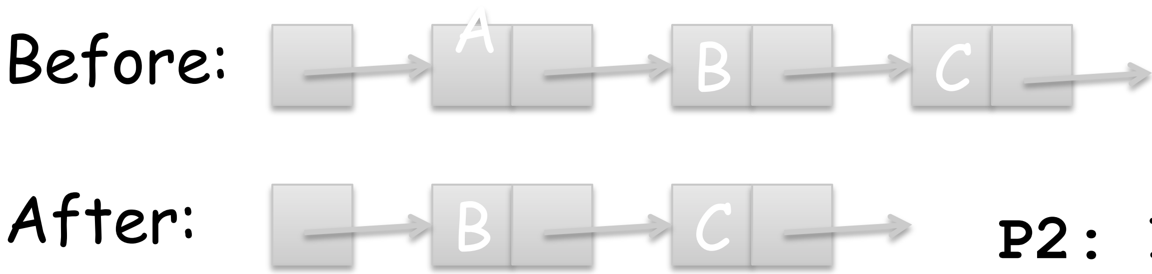
Concurrency : Stack Example 2

```
struct stack {
  link* headPtr;

  void push(link* a) {
    do {
      link* h = headPtr;
      a->next = h;
    while (!CAS(&headPtr, h, a)); }

  link* pop() {
    do {
      link* h = headPtr;
      if (h == NULL) return NULL;
      link* nxt = h->next;
    while (!CAS(&headPtr, h, nxt))}
    return h;}
}
```



Concurrency : Stack Example 2'

```
P1 : x = s.pop();  y = s.pop();  s.push(x);
P2 : z = s.pop();
```




```
P2: h = headPtr;
P2: nxt = h->next;
P1: everything
P2: CAS(&headPtr,h,nxt)
```

The ABA problem

Can be fixed with counter and 2CAS, but…

Concurrency : Stack Example 3

```
struct link {int v; link* next;}

struct stack {
  link* headPtr;

  void push(link* a) {
    atomic {
      a->next = headPtr;
      headPtr = a;   }}

  link* pop() {
    atomic {
      link* h = headPtr;
      if (headPtr != NULL)
        headPtr = headPtr->next;
      return h;}}
}
```

Concurrency : Stack Example 3'

```
void swapTop(stack s) {
  link* x = s.pop();
  link* y = s.pop();
  push(x);
  push(y);
}
```

Queues are trickier than stacks.

Styles of Parallel Programming

- Data parallelism/Bulk Synchronous/SPMD
- Nested parallelism : what we covered
- Message passing
- Futures (other pipelined parallelism)
- General Concurrency

Futures: Example

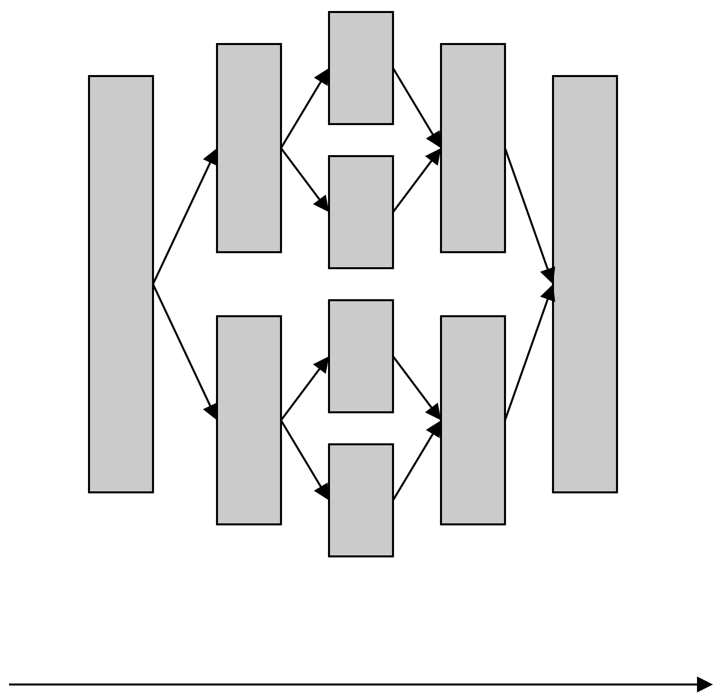
```
fun quickSort S = let
  fun qs([], rest) = rest
    | qs(h::T, rest) =
      let
        val L1 = filter (fn b => b < a) T
        val L2 = filter (fn b => b >= a) T
      in
        qs(L1,(a::(qs(L2, rest)))
      end

  fun filter(f,[]) = []
    | filter(f,h::T) =
       if f(h) then(h::filter(f,T))
       else filter(f,T)
in
  qs(S,[])
end
```

Futures: Example

```
fun quickSort S = let
  fun qs([], rest) = rest
    | qs(h::T, rest) =
      let
        val L1 = filter (fn b => b < a) T
        val L2 = filter (fn b => b >= a) T
      in
        qs(L1, future(a::(qs(L2, rest)))
      end

  fun filter(f,[]) = []
    | filter(f,h::T) =
       if f(h) then future(h::filter(f,T))
       else filter(f,T)
in
  qs(S,[])
end
```

Quicksort: Nested Parallel

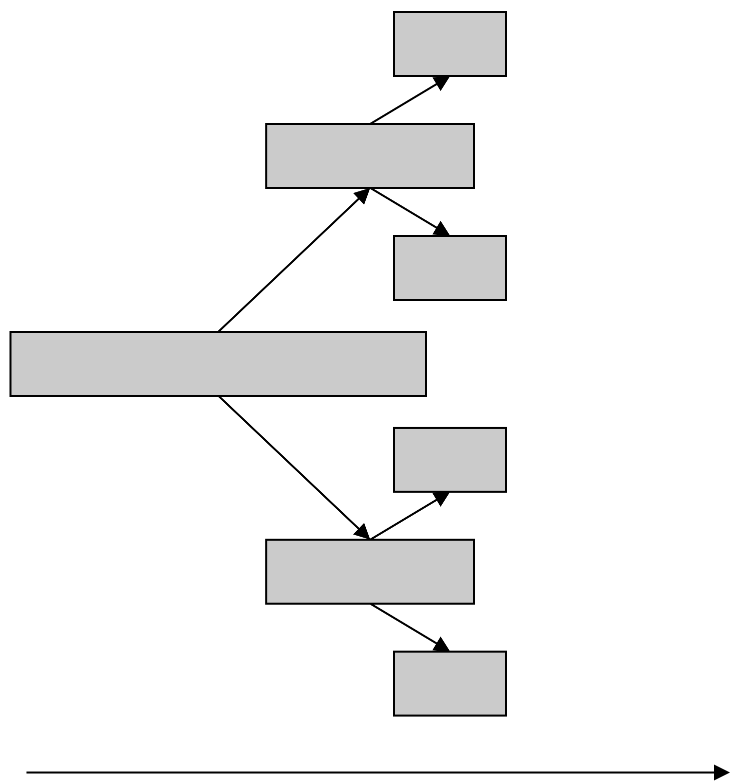
Parallel Partition and Append



Work = $O(n \log n)$

Span = $O(\lg^2 n)$

Quicksort: Futures



Work = O(n log n)

Span = O(n)

Styles of Parallel Programming

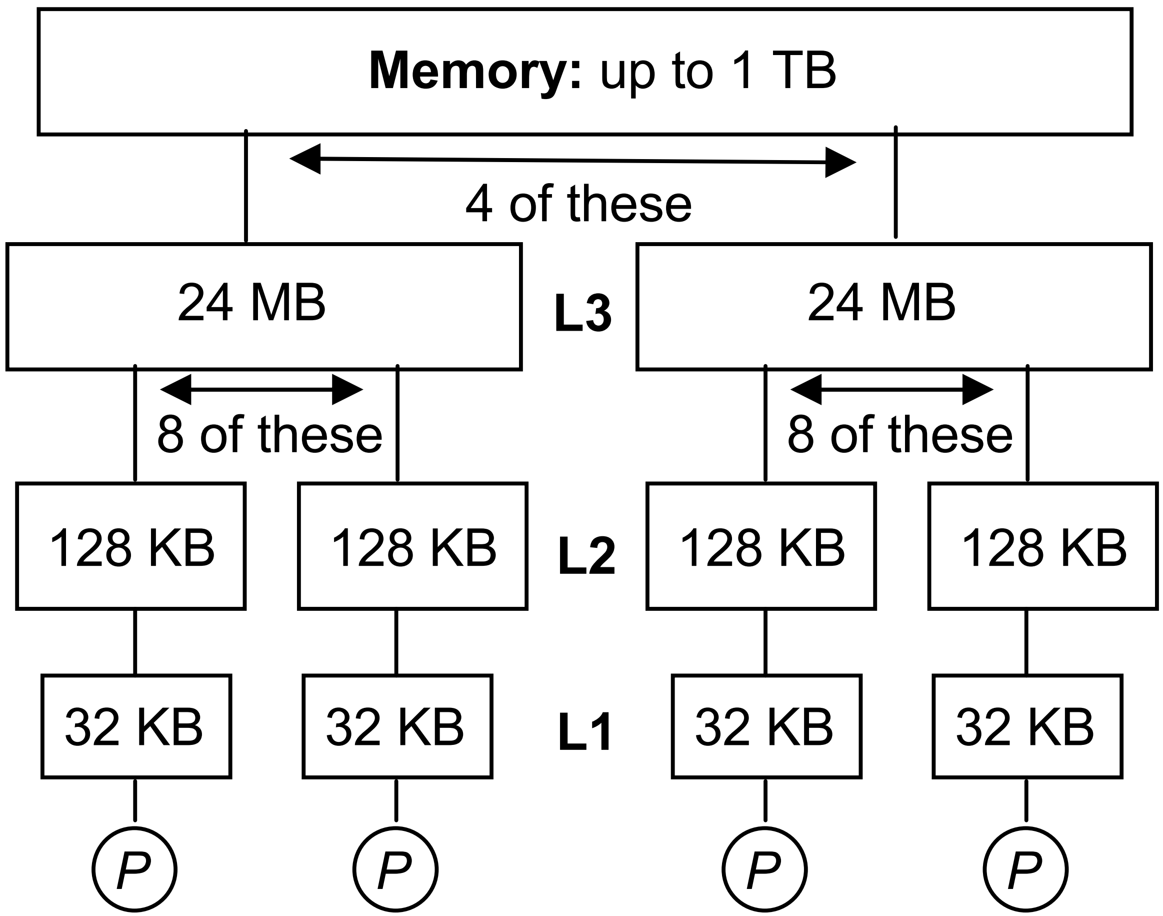
Data parallelism/Bulk Synchronous/SPMD

* Nested parallelism : what we covered

Message passing

* Futures (other pipelined parallelism)
* General Concurrency

Xeon 7500



Memory: up to 1 TB
4 of these
24 MB
L3
24 MB
8 of these
8 of these
128 KB
128 KB
L2
128 KB
128 KB
32 KB
32 KB
L1
32 KB
32 KB
P
P
P
P

Question

How do we get nested parallel programs to work well on a fixed set of processors? Programs say nothing about processors.

Answer: good schedulers

Greedy Schedules

"Speedup versus Efficiency in Parallel Systems",
Eager, Zahorjan and Lazowska, 1989

For any greedy schedule:

Efficiency = $\dfrac{W}{T_P} \geq \dfrac{PW}{W + D(P-1)}$

Parallel Time = $T_P \leq \dfrac{W}{P} + D$

Types of Schedulers

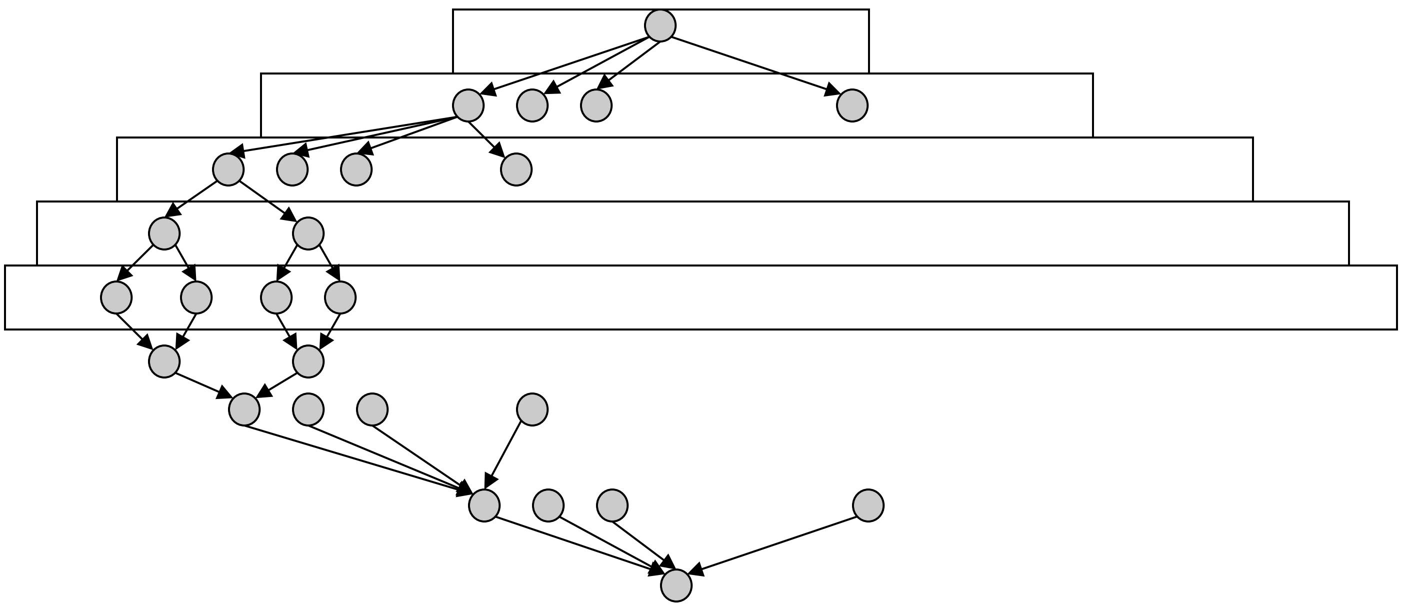
Bread-First Schedulers

Work-stealing Schedulers

Depth-first Schedulers

Breadth First Schedules

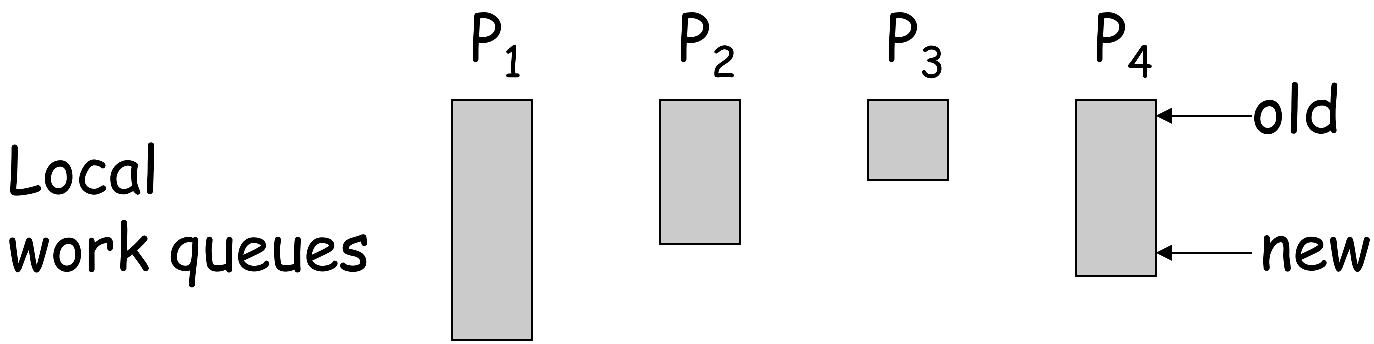
Most naïve schedule. Used by most implementations of P-threads.



$O(n^3)$ tasks

Bad space usage, bad locality

Work Stealing



push new jobs on "new" end

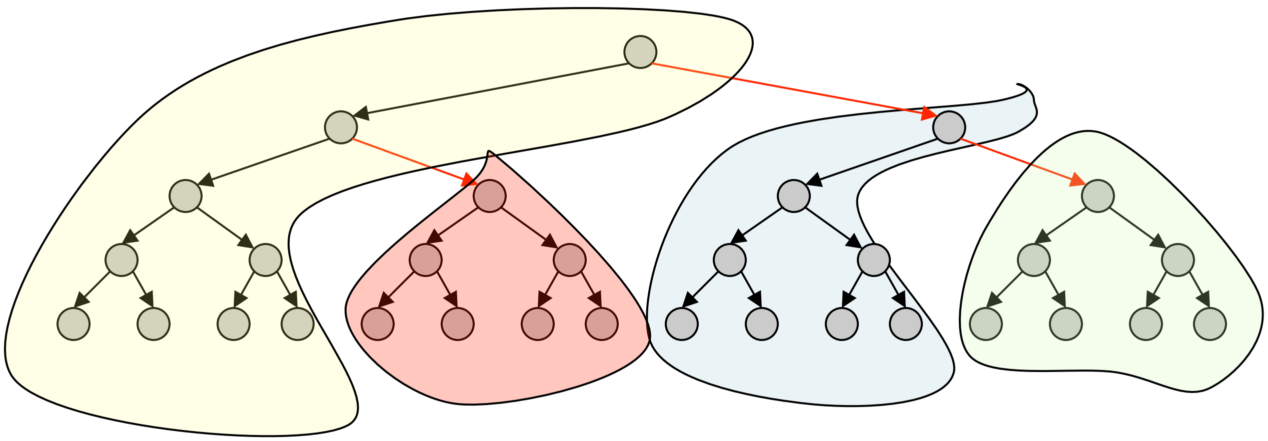
pop jobs from "new" end

If processor runs out of work, then "**steal**" from another "old" end

Each processor tends to execute a sequential part of the computation.

Work Stealing

Tends to schedule "sequential blocks" of tasks



 = steal

Work Stealing Theory

For strict computations

Blumofe and Leiserson, 1999

# of steals = $O(PD)$

Space = $O(PS_1)$ S_1 is the sequential space

Acar, Blelloch and Blumofe, 2003

# of cache misses on distributed caches

$M_1 + O(CPD)$

M_1 = sequential misses, C = cache size

Work Stealing Practice

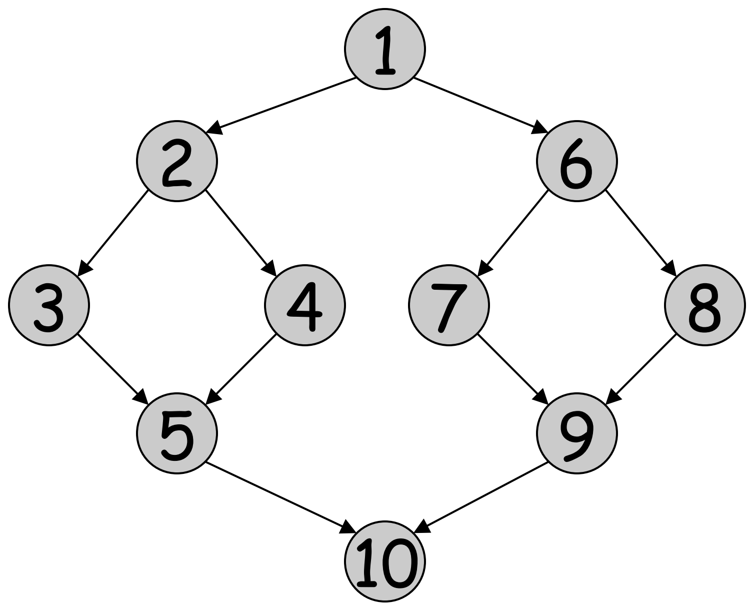
Used in Cilk Scheduler

- Small overheads because common case of pushing/popping from local queue can be made fast (with good data structures and compiler help).
- No contention on a global queue
- Has good distributed cache behavior
- Can indeed require $O(S_1 P)$ memory

Used in X10 scheduler, and others

Parallel Depth First Schedules (P-DFS)

List scheduling based on Depth-First ordering



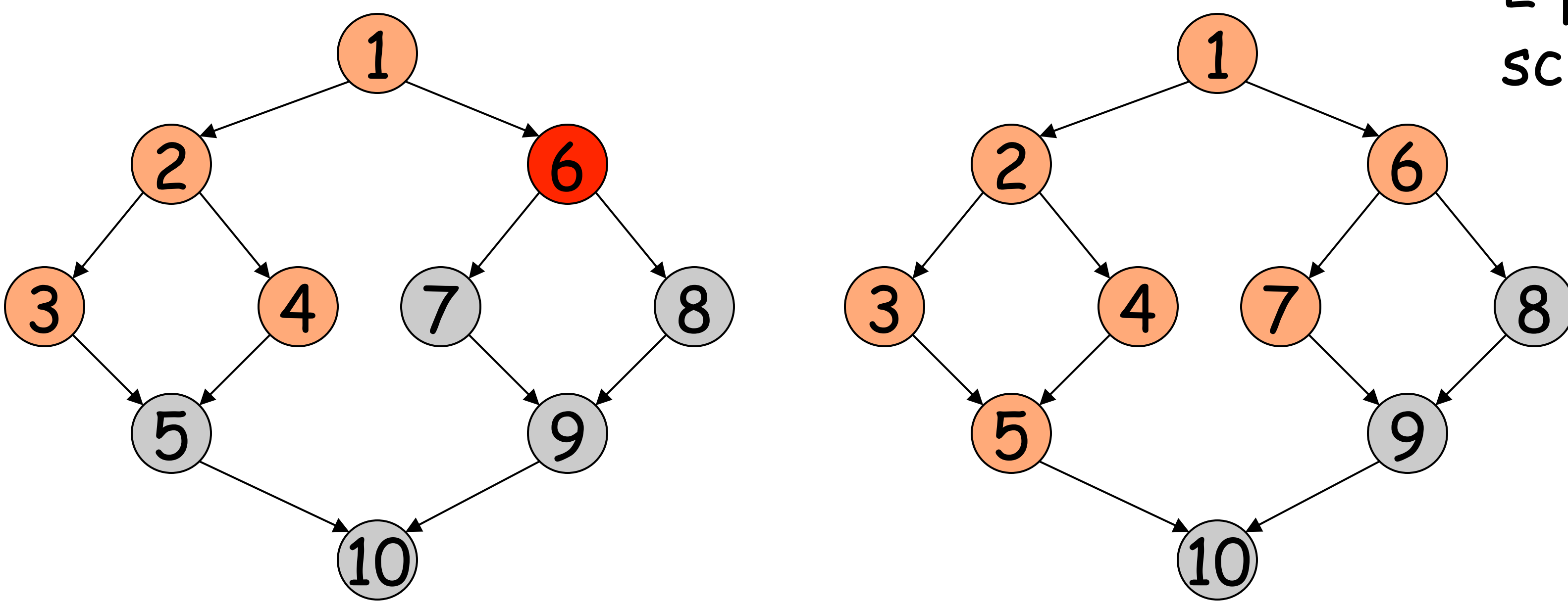
2 processor schedule

1
2, 6
3, 4
5, 7
8
9
10

For strict computations a shared stack implements a P-DFS

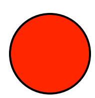
“Premature task” in P-DFS

A running task is premature if there is an earlier sequential task that is not complete



2 processor schedule

1
2, 6
3, 4
5, 7
8
9
10

 = premature

P-DFS Theory

Blelloch, Gibbons, Matias, 1999

For any computation:

Premature nodes at any time = $O(PD)$

Space = $S_1 + O(PD)$

Blelloch and Gibbons, 2004

With a shared cache of size $C_1 + O(PD)$ we have $M_p = M_1$

P-DFS Practice

Experimentally uses less memory than work stealing and performs better on a shared cache.

Requires some "coarsening" to reduce overheads

Conclusions

- lots of parallel languages
- lots of parallel programming styles
- high-level ideas cross between languages and styles
- scheduling is important