Multithreaded Parallelism on Multicore Architectures

Marc Moreno Maza

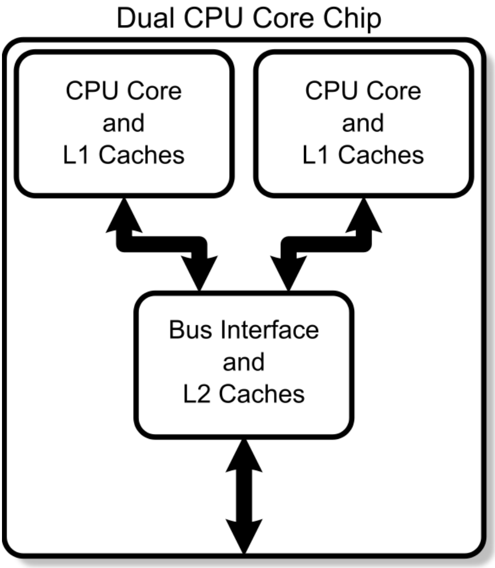
University of Western Ontario, Canada

CS2101 March 2012

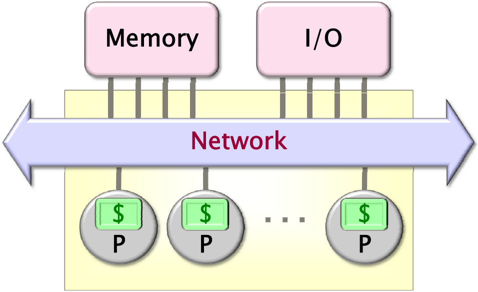
Plan

1. Multicore programming
    - Multicore architectures
2. Cilk / Cilk++ / Cilk Plus
3. The fork-join multithreaded programming model
4. Practical issues and optimization tricks

Plan



- A multi-core processor is an integrated circuit to which two or more individual processors (called cores in this sense) have been attached.



Chip Multiprocessor (CMP)

- Cores on a multi-core device can be coupled tightly or loosely:
  - may share or may not share a cache,
  - implement inter-core communications methods or message passing.
- Cores on a multi-core implement the same architecture features as single-core systems such as instruction pipeline parallelism (ILP), vector-processing, SIMD or multi-threading.

Cache Coherence (1/6)

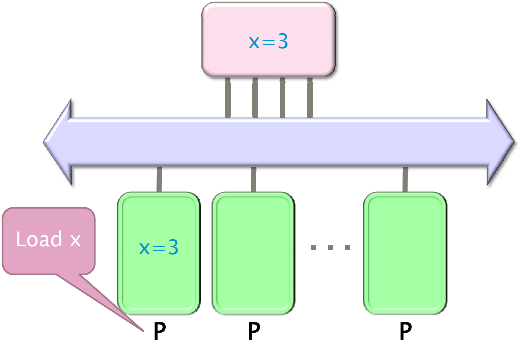


Figure: Processor P_1 reads x=3 first from the backing store (higher-level memory)

Cache Coherence (2/6)

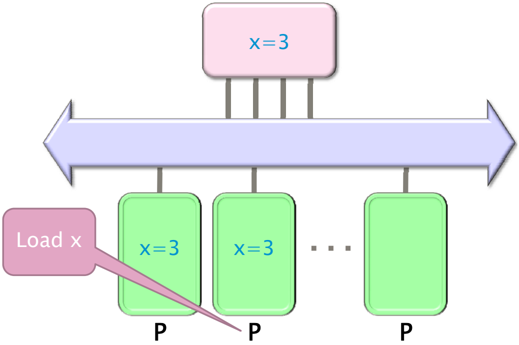


Figure: Next, Processor P_2 loads x=3 from the same memory

Cache Coherence (3/6)

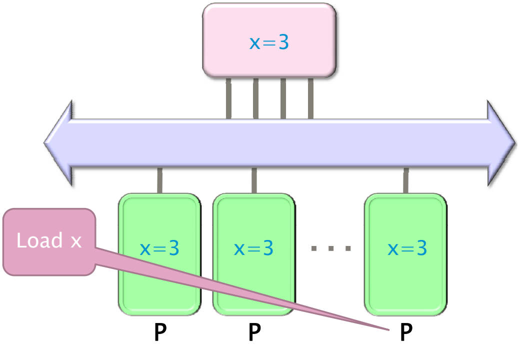


Figure: Processor P_4 loads x=3 from the same memory

Cache Coherence (4/6)

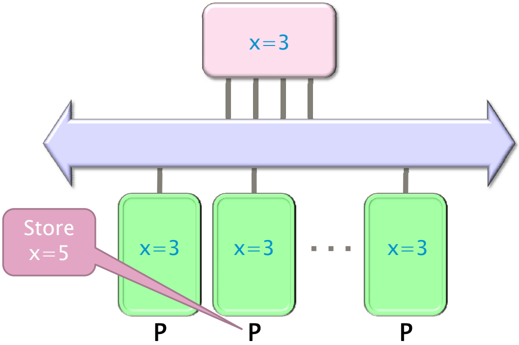


Figure: Processor P_2 issues a write x=5

Cache Coherence (5/6)

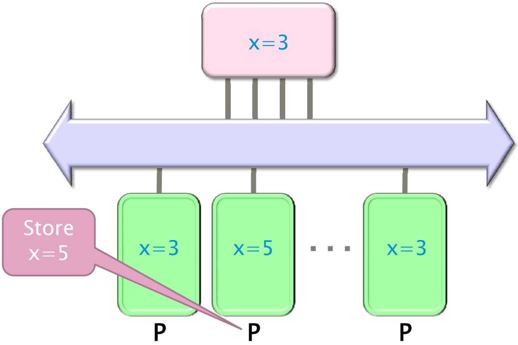


Figure: Processor P_2 writes x=5 in his local cache

Cache Coherence (6/6)

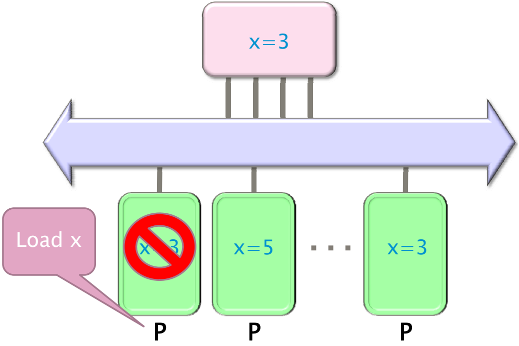


Figure: Processor P_1 issues a read `x`, which is now invalid in its cache

MSI Protocol

- In this cache coherence protocol each block contained inside a cache can have one of three possible states:
  - **M:** the cache line has been **modified** and the corresponding data is inconsistent with the backing store; the cache has the responsibility to write the block to the backing store when it is evicted.
  - **S:** this block is unmodified and is **shared**, that is, exists in at least one cache. The cache can evict the data without writing it to the backing store.
  - **I:** this block is **invalid**, and must be fetched from memory or another cache if the block is to be stored in this cache.
- These coherency states are maintained through communication between the caches and the backing store.
- The caches have different responsibilities when blocks are read or written, or when they learn of other caches issuing reads or writes for a block.

True Sharing and False Sharing

- **True sharing:**
  - True sharing cache misses occur whenever two processors access the same data word
  - True sharing requires the processors involved to explicitly synchronize with each other to ensure program correctness.
  - A computation is said to have **temporal locality** if it re-uses much of the data it has been accessing.
  - Programs with high temporal locality tend to have less true sharing.
- **False sharing:**
  - False sharing results when different processors use different data that happen to be co-located on the same cache line
  - A computation is said to have **spatial locality** if it uses multiple words in a cache line before the line is displaced from the cache
  - Enhancing spatial locality often minimizes false sharing
- See *Data and Computation Transformations for Multiprocessors* by J.M. Anderson, S.P. Amarasinghe and M.S. Lam `http://suif.stanford.edu/papers/anderson95/paper.html`

Multi-core processor (cntd)

- **Advantages:**
  - Cache coherency circuitry operate at higher rate than off-chip.
  - Reduced power consumption for a dual core vs two coupled single-core processors (better quality communication signals, cache can be shared)
- **Challenges:**
  - Adjustments to existing software (including OS) are required to maximize performance
  - Production yields down (an Intel quad-core is in fact a double dual-core)
  - Two processing cores sharing the same bus and memory bandwidth may limit performances
  - High levels of false or true sharing and synchronization can easily overwhelm the advantage of parallelism

From Cilk to Cilk++ and Cilk Plus

- `Cilk` has been developed since 1994 at the MIT Laboratory for Computer Science by Prof. Charles E. Leiserson and his group, in particular by Matteo Frigo.
- Besides being used for research and teaching, `Cilk` was the system used to code the three world-class chess programs: Tech, Socrates, and Cilkchess.
- Over the years, the implementations of `Cilk` have run on computers ranging from networks of Linux laptops to an 1824-nodes Intel Paragon.
- From 2007 to 2009 Cilk has lead to `Cilk++`, developed by Cilk Arts, an MIT spin-off, which was acquired by Intel in July 2009 and became Cilk Plus, see `http://www.cilk.com/`
- `Cilk++` can be freely downloaded at `http://software.intel.com/en-us/articles/download-intel-ci`
- `Cilk` is still developed at MIT `http://supertech.csail.mit.edu/cilk/`

Cilk++ (and Cilk Plus)

- `Cilk++` (resp. `Cilk`) is a small set of linguistic extensions to C++ (resp. `C`) supporting fork-join parallelism
- Both `Cilk` and `Cilk++` feature a provably efficient work-stealing scheduler.
- `Cilk++` provides a hyperobject library for parallelizing code with global variables and performing reduction for data aggregation.
- `Cilk++` includes the `Cilkscreen` race detector and the `Cilkview` performance analyzer.

Nested Parallelism in Cilk ++

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

- The named child function `cilk_spawn fib(n-1)` may execute in parallel with its parent
- Cilk++ keywords `cilk_spawn` and `cilk_sync` grant permissions for parallel execution. They do not command parallel execution.

Loop Parallelism in Cilk ++

$$\underset{A}{\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}} \Rightarrow \underset{A^T}{\begin{pmatrix} a_{11} & a_{21} & \cdots & a_{n1} \\ a_{12} & a_{22} & \cdots & a_{n2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1n} & a_{2n} & \cdots & a_{nn} \end{pmatrix}}$$

```
// indices run from 0, not 1
cilk_for (int i=1; i<n; ++i) {
    for (int j=0; j<i; ++j) {
        double temp = A[i][j];
        A[i][j] = A[j][i];
        A[j][i] = temp;
    }
}
```

The iterations of a `cilk_for` loop may execute in parallel.

Serial Semantics (1/2)

- `Cilk` (resp. `Cilk++`) is a multithreaded language for parallel programming that generalizes the semantics of `C` (resp. `C++`) by introducing linguistic constructs for parallel control.
- `Cilk` (resp. `Cilk++`) is a faithful extension of `C` (resp. `C++`):
  - The `C` (resp. `C++`) elision of a `Cilk` (resp. `Cilk++`) is a correct implementation of the semantics of the program.
  - Moreover, on one processor, a parallel `Cilk` (resp. `Cilk++`) program scales down to run nearly as fast as its `C` (resp. `C++`) elision.
- To obtain the serialization of a `Cilk++` program

```
#define cilk_for for
#define cilk_spawn
#define cilk_sync
```

Serial Semantics (2/2)

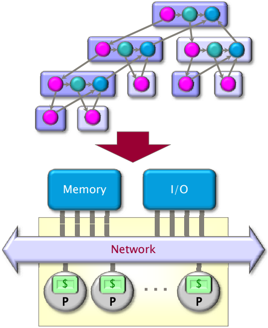
```
int fib (int n) {
  if (n<2) return (n);
  else {
    int x,y;
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
    cilk_sync;
    return (x+y);
  }
}
```

Cilk++ source

```
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

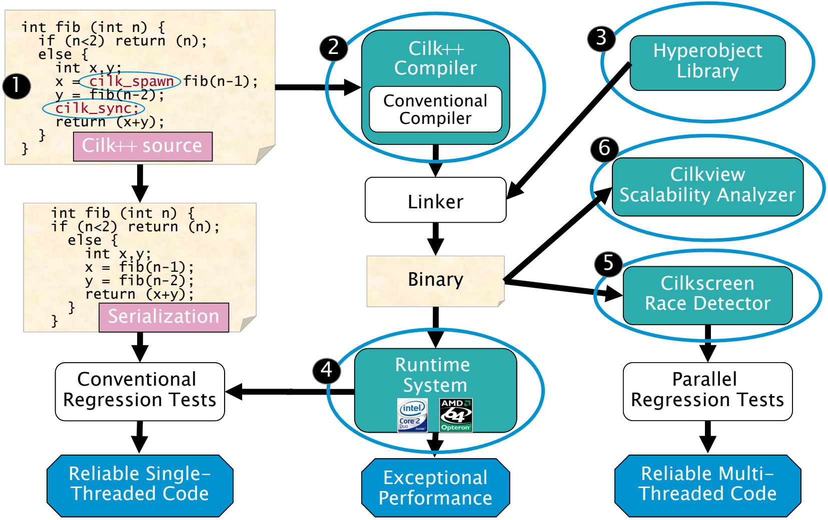
Scheduling



A **scheduler**'s job is to map a computation to particular processors. Such a mapping is called a **schedule**.

- If decisions are made at runtime, the scheduler is *online*, otherwise, it is *offline*
- `Cilk++`'s scheduler maps strands onto processors dynamically at runtime.

The Cilk++ Platform



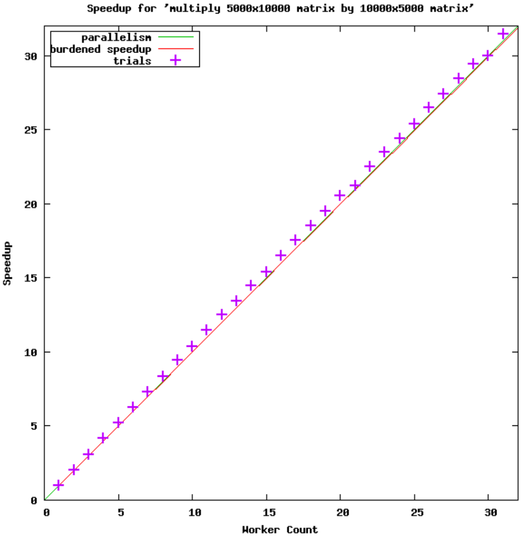
Benchmarks for the parallel version of the cache-oblivious mm

Multiplying a 4000x8000 matrix by a 8000x4000 matrix

- on 32 cores = 8 sockets x 4 cores (Quad Core AMD Opteron 8354) per socket.
- The 32 cores share a L3 32-way set-associative cache of 2 Mbytes.

#core	Elision (s)	Parallel (s)	speedup
8	420.906	51.365	8.19
16	432.419	25.845	16.73
24	413.681	17.361	23.83
32	389.300	13.051	29.83

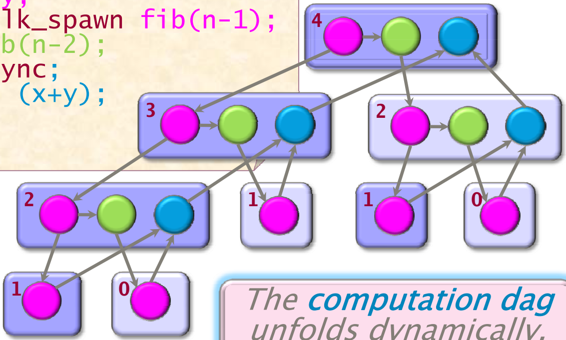
So does the (tuned) cache-oblivious matrix multiplication



The fork-join parallelism model

```
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);
  }
}
```

Example:
`fib(4)`

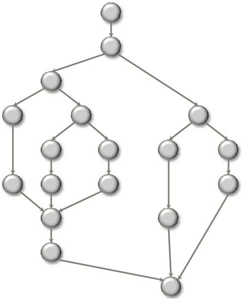


"Processor oblivious"

*The **computation dag** unfolds dynamically.*

We shall also call this model **multithreaded parallelism**.

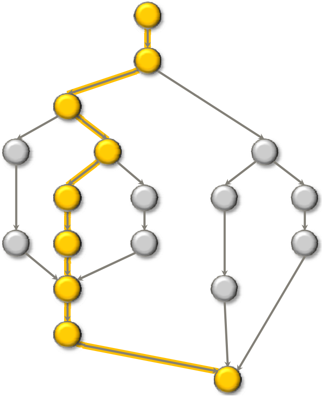
Work and span



We define several performance measures. We assume an ideal situation: no cache issues, no interprocessor costs:

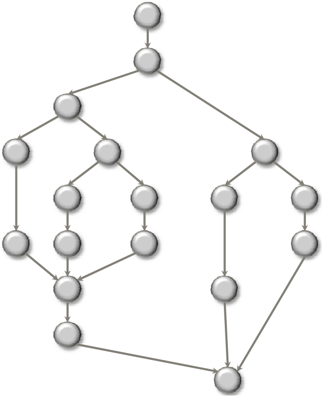
- T_p is the minimum running time on p processors
- T_1 is called the **work**, that is, the sum of the number of instructions at each node.
- T_∞ is the minimum running time with infinitely many processors, called the **span**

The critical path length



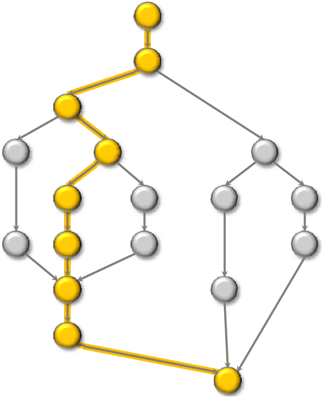
Assuming all strands run in unit time, the longest path in the DAG is equal to T_∞. For this reason, T_∞ is also referred to as the **critical path length**.

Work law



- We have: $T_p \geq T_1/p$.
- Indeed, in the best case, p processors can do p works per unit of time.

Span law



- We have: $T_p \geq T_\infty$.
- Indeed, $T_p < T_\infty$ contradicts the definitions of T_p and T_∞.

Speedup on p processors

- T_1/T_p is called the **speedup on p processors**
- A parallel program execution can have:
  - **linear speedup**: $T_1/T_P = \Theta(p)$
  - **superlinear speedup**: $T_1/T_P = \omega(p)$ (not possible in this model, though it is possible in others)
  - **sublinear speedup**: $T_1/T_P = o(p)$

For loop parallelism in Cilk++

$$\underset{A}{\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}} \Rightarrow \underset{A^T}{\begin{pmatrix} a_{11} & a_{21} & \cdots & a_{n1} \\ a_{12} & a_{22} & \cdots & a_{n2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1n} & a_{2n} & \cdots & a_{nn} \end{pmatrix}}$$

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

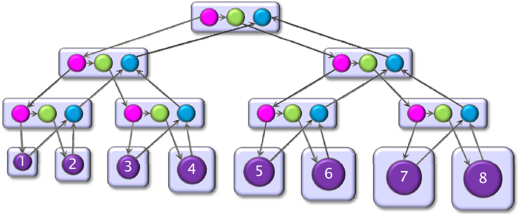
The iterations of a `cilk_for` loop execute in parallel.

Implementation of `for` loops in `Cilk++`

Up to details (next week!) the previous loop is compiled as follows, using a **divide-and-conquer implementation**:

```
void recur(int lo, int hi) {
    if (hi > lo) { // coarsen
        int mid = lo + (hi - lo)/2;
        cilk_spawn recur(lo, mid);
        recur(mid, hi);
        cilk_sync;
    } else
        for (int j=0; j<i; ++j) {
            double temp = A[i][j];
            A[i][j] = A[j][i];
            A[j][i] = temp;
        }
    }
}
```

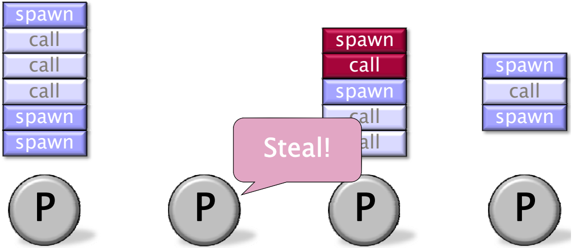
Analysis of parallel `for` loops



Here we do not assume that each strand runs in unit time.

- **Span of loop control**: $\Theta(\log(n))$
- **Max span of an iteration**: $\Theta(n)$
- **Span**: $\Theta(n)$
- **Work**: $\Theta(n^2)$
- **Parallelism**: $\Theta(n)$

The work-stealing scheduler



Performances of the work-stealing scheduler

Assume that

- each strand executes in unit time,
- for almost all "parallel steps" there are at least p strands to run,
- each processor is either working or stealing.

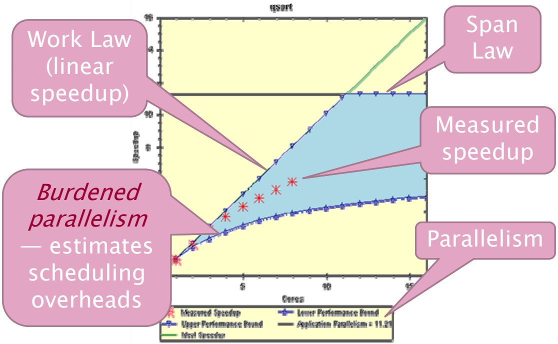
Then, the randomized work-stealing scheduler is expected to run in

$$T_P = T_1/p + O(T_\infty)$$

Overheads and burden

- Many factors (simplification assumptions of the fork-join parallelism model, architecture limitation, costs of executing the parallel constructs, overheads of scheduling) will make T_p smaller in practice than $T_1/p + T_\infty$.
- One may want to estimate the impact of those factors:
  1. by improving the estimate of the *randomized work-stealing complexity result*
  2. by comparing a `Cilk++` program with its `C++` elision
  3. by estimating the costs of spawning and synchronizing
- `Cilk++` estimates T_p as $T_p = T_1/p + 1.7 \text{ burden\_span}$, where burden_span is 15000 instructions times the number of continuation edges along the critical path.

Cilkview



- `Cilkview` computes work and span to derive upper bounds on parallel performance
- `Cilkview` also estimates scheduling overhead to compute a burdened span for lower bounds.

The Fibonacci Cilk++ example

Code fragment

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

Fibonacci program timing

The environment for benchmarking:

- model name : Intel(R) Core(TM)2 Quad CPU Q6600 @ 2.40GHz
- L2 cache size : 4096 KB
- memory size : 3 GB

	#cores = 1	#cores = 2		#cores = 4	
n	timing(s)	timing(s)	speedup	timing(s)	speedup
30	0.086	0.046	1.870	0.025	3.440
35	0.776	0.436	1.780	0.206	3.767
40	8.931	4.842	1.844	2.399	3.723
45	105.263	54.017	1.949	27.200	3.870
50	1165.000	665.115	1.752	340.638	3.420

Quicksort

code in cilk/examples/qsort

```
void sample_qsort(int * begin, int * end)
{
   if (begin != end) {
         --end;
        int * middle = std::partition(begin, end,
            std::bind2nd(std::less<int>(), *end));
        using std::swap;
        swap(*end, *middle);
        cilk_spawn sample_qsort(begin, middle);
        sample_qsort(++middle, ++end);
        cilk_sync;
    }
}
```

Quicksort timing

Timing for sorting an array of integers:

	#cores = 1	#cores = 2		#cores = 4	
# of int	timing(s)	timing(s)	speedup	timing(s)	speedup
10×10^6	1.958	1.016	1.927	0.541	3.619
50×10^6	10.518	5.469	1.923	2.847	3.694
100×10^6	21.481	11.096	1.936	5.954	3.608
500×10^6	114.300	57.996	1.971	31.086	3.677

Matrix multiplication

Code in cilk/examples/matrix

Timing of multiplying a 687×837 matrix by a 837×1107 matrix

	iterative			recursive		
threshold	st(s)	pt(s)	su	st(s)	pt (s)	su
10	1.273	1.165	0.721	1.674	0.399	4.195
16	1.270	1.787	0.711	1.408	0.349	4.034
32	1.280	1.757	0.729	1.223	0.308	3.971
48	1.258	1.760	0.715	1.164	0.293	3.973
64	1.258	1.798	0.700	1.159	0.291	3.983
80	1.252	1.773	0.706	1.267	0.320	3.959

st = sequential time; pt = parallel time with 4 cores; su = speedup

The `cilkview` example from the documentation

Using cilk_for to perform operations over an array in parallel:

```
static const int COUNT = 4;
static const int ITERATION = 1000000;
long arr[COUNT];
long do_work(long k){
  long x = 15;
  static const int nn = 87;
  for (long i = 1; i < nn; ++i)
    x = x / i + k % i;
  return x;
}
int cilk_main(){
  for (int j = 0; j < ITERATION; j++)
    cilk_for (int i = 0; i < COUNT; i++)
      arr[i] += do_work( j * i + i + j);
}
```

```
1) Parallelism Profile
   Work :                                  6,480,801,250 ins
   Span :                                  2,116,801,250 ins
   Burdened span :                         31,920,801,250 ins
   Parallelism :                                   3.06
   Burdened parallelism :                          0.20
   Number of spawns/syncs:                         3,000,000
   Average instructions / strand :                 720
   Strands along span :                            4,000,001
   Average instructions / strand on span : 529
2) Speedup Estimate
   2 processors:          0.21 - 2.00
   4 processors:          0.15 - 3.06
   8 processors:          0.13 - 3.06
   16 processors:         0.13 - 3.06
   32 processors:         0.12 - 3.06
```

A simple fix

Inverting the two for loops

```
int cilk_main()
{
  cilk_for (int i = 0; i < COUNT; i++)
    for (int j = 0; j < ITERATION; j++)
      arr[i] += do_work( j * i + i + j);
}
```

```
1) Parallelism Profile
   Work :                                   5,295,801,529 ins
   Span :                                   1,326,801,107 ins
   Burdened span :                          1,326,830,911 ins
   Parallelism :                                   3.99
   Burdened parallelism :                          3.99
   Number of spawns/syncs:                         3
   Average instructions / strand :                 529,580,152
   Strands along span :                            5
   Average instructions / strand on span: 265,360,221
2) Speedup Estimate
   2 processors:          1.40 - 2.00
   4 processors:          1.76 - 3.99
   8 processors:          2.01 - 3.99
   16 processors:         2.17 - 3.99
   32 processors:         2.25 - 3.99
```

Timing

	`#cores = 1`	`#cores = 2`		`#cores = 4`	
version	timing(s)	timing(s)	speedup	timing(s)	speedup
original	7.719	9.611	0.803	10.758	0.718
improved	7.471	3.724	2.006	1.888	3.957

Example 1: a small loop with grain size = 1

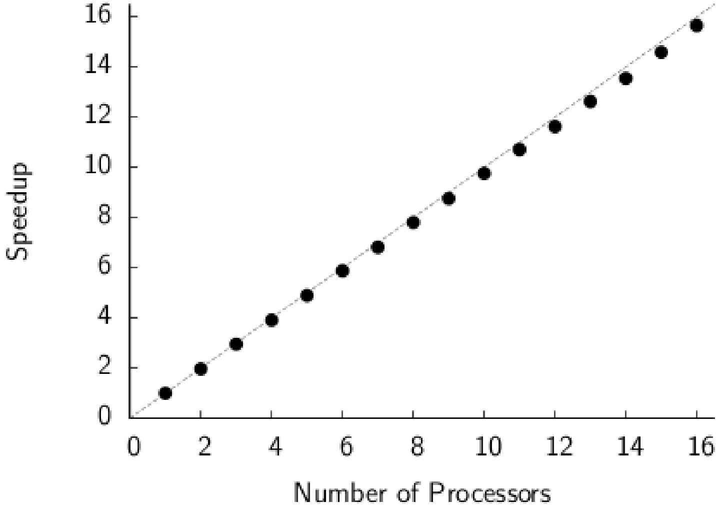
Code:

```
const int N = 100 * 1000 * 1000;

void cilk_for_grainsize_1()
{
#pragma cilk_grainsize = 1
    cilk_for (int i = 0; i < N; ++i)
        fib(2);
}
```

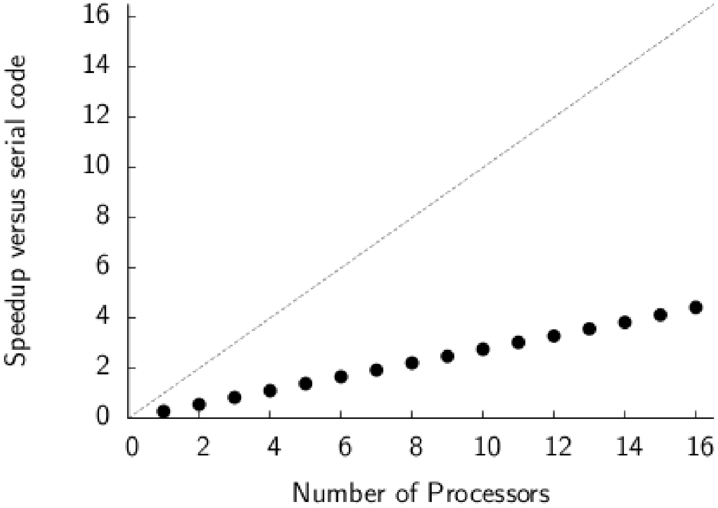
Expectations:

- Parallelism should be large, perhaps $\Theta(N)$ or $\Theta(N/\log N)$.
- We should see great speedup.

Speedup is indeed great. . .



. . . but performance is lousy



Recall how `cilk_for` is implemented

Source:

```
cilk_for (int i = A; i < B; ++i)
     BODY(i)
```

Implementation:

```
void recur(int lo, int hi) {
     if ((hi - lo) > GRAINSIZE) {
          int mid = lo + (hi - lo) / 2;
          cilk_spawn recur(lo, mid);
          cilk_spawn recur(mid, hi);
     } else
          for (int i = lo; i < hi; ++i)
               BODY(i);
}

recur(A, B);
```

Default grain size

Cilk++ chooses a grain size if you don't specify one.

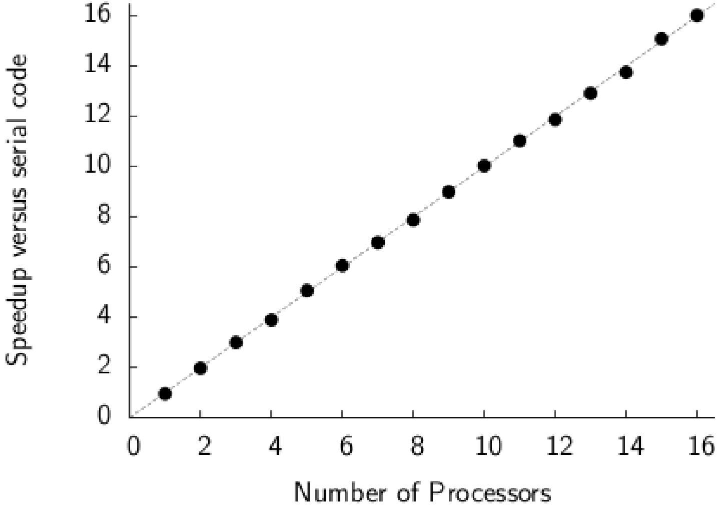
```
void cilk_for_default_grainsize()
{
    cilk_for (int i = 0; i < N; ++i)
        fib(2);
}
```

Cilk++'s heuristic for the grain size:

$$\text{grain size} = \min\left\{\frac{N}{8P}, 512\right\} .$$

- Generates about $8P$ parallel leaves.
- Works well if the loop iterations are not too unbalanced.

Speedup with default grain size



Large grain size

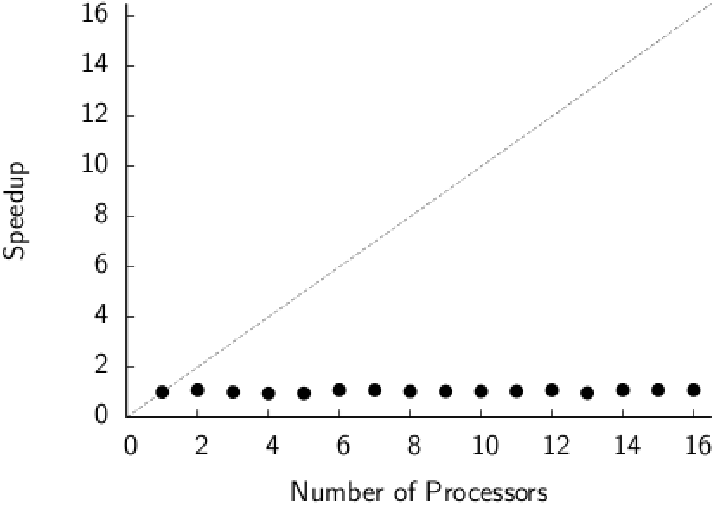
A large grain size should be even faster, right?

```
void cilk_for_large_grainsize()
{
#pragma cilk_grainsize = N
    cilk_for (int i = 0; i < N; ++i)
        fib(2);
}
```

Actually, no (except for noise):

Grain size	Runtime
1	8.55 s
default (= 512)	2.44 s
N $(= 10^8)$	2.42 s

Speedup with grain size = N



Tradeoff between grain size and parallelism

Use the PPA to understand the tradeoff:

Grain size	Parallelism
1	6,951,154
default $(= 512)$	248,784
$N\ (= 10^8)$	1

In the PPA, $P = 1$:

$$\text{default grain size} = \min\left\{\frac{N}{8P}, 512\right\} = \min\left\{\frac{N}{8}, 512\right\}\ .$$

Lessons learned

- Measure overhead before measuring speedup.
  - Compare 1-processor Cilk++ versus serial code.
- Small grain size $\Rightarrow$ higher work overhead.
- Large grain size $\Rightarrow$ less parallelism.
- The default grain size is designed for small loops that are reasonably balanced.
  - You may want to use a smaller grain size for unbalanced loops or loops with large bodies.
- Use the PPA to measure the parallelism of your program.

Example 2: A `for` loop that spawns

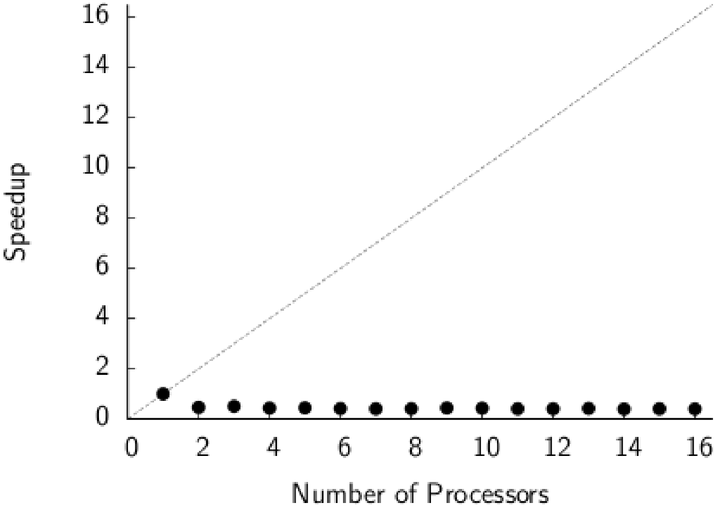
Code:

```
const int N = 10 * 1000 * 1000;

/* empty test function */
void f() { }

void for_spawn()
{
    for (int i = 0; i < N; ++i)
        cilk_spawn f();
}
```

Expectations:

- I am spawning N parallel things.
- Parallelism should be $\Theta(N)$, right?

“Speedup” of `for_spawn()`



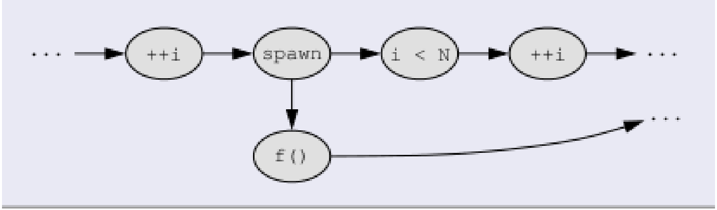
Insufficient parallelism

PPA analysis:

- PPA says that both work and span are $\Theta(N)$.
- Parallelism is ≈ 1.62, independent of N.
- Too little parallelism: no speedup.

Why is the span $\Theta(N)$?

```
for (int i = 0; i < N; ++i)
    cilk_spawn f();
```



Alternative: a cilk_for loop.

Code:

```
/* empty test function */
void f() { }

void test_cilk_for()
{
    cilk_for (int i = 0; i < N; ++i)
        f();
}
```

PPA analysis:

The parallelism is about 2000 (with default grain size).

- The parallelism is high.
- As we saw earlier, this kind of loop yields good performance and speedup.

Lessons learned

- `cilk_for()` is different from `for(...) cilk_spawn`.
- The span of `for(...) cilk_spawn` is $\Omega(N)$.
- For simple flat loops, `cilk_for()` is generally preferable because it has higher parallelism.
- (However, `for(...) cilk_spawn` might be better for recursively nested loops.)
- Use the PPA to measure the parallelism of your program.

Example 3: Vector addition

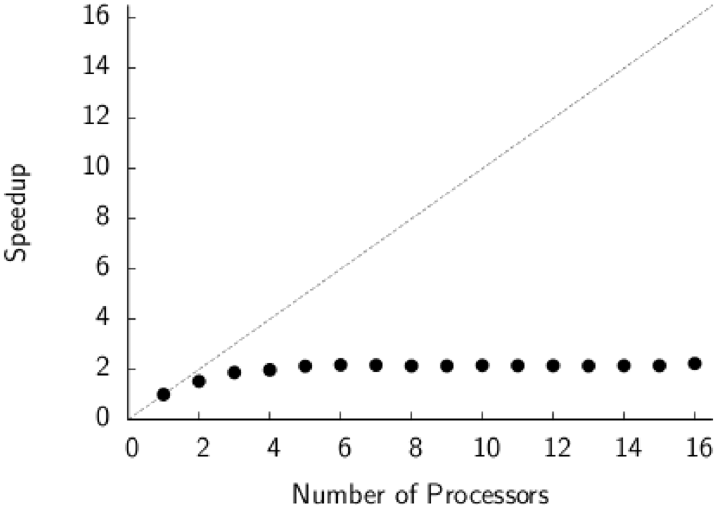
Code:

```
const int N = 50 * 1000 * 1000;

double A[N], B[N], C[N];

void vector_add()
{
    cilk_for (int i = 0; i < N; ++i)
        A[i] = B[i] + C[i];
}
```

Expectations:

- The PPA says that the parallelism is 68,377.
- This will work great!

Speedup of `vector_add()`



Bandwidth of the memory system

A typical machine: AMD Phenom 920 (Feb. 2009).

Cache level	daxpy bandwidth
L1	19.6 GB/s per core
L2	18.3 GB/s per core
L3	13.8 GB/s shared
DRAM	7.1 GB/s shared

daxpy: `x[i] = a*x[i] + y[i]`, double precision.

The memory bottleneck:

- A single core can generally saturate most of the memory hierarchy.
- Multiple cores that access memory will conflict and slow each other down.

How do you determine if memory is a bottleneck?

Hard problem:

- No general solution.
- Requires guesswork.

Two useful techniques:

- Use a profiler such as the Intel VTune.
  - Interpreting the output is nontrivial.
  - No sensitivity analysis.
- Perturb the environment to understand the effect of the CPU and memory speeds upon the program speed.

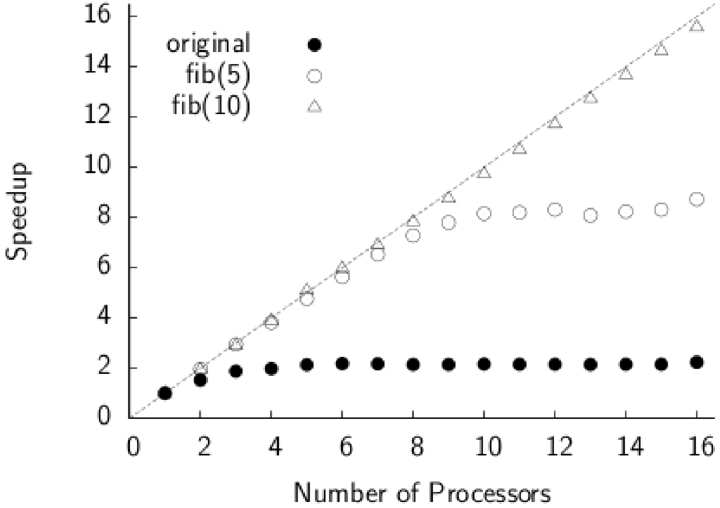
How to perturb the environment

- Overclock/underclock the processor, e.g. using the power controls.
  - If the program runs at the same speed on a slower processor, then the memory is (probably) a bottleneck.
- Overclock/underclock the DRAM from the BIOS.
  - If the program runs at the same speed on a slower DRAM, then the memory is not a bottleneck.
- Add spurious work to your program while keeping the memory accesses constant.
- Run P independent copies of the serial program concurrently.
  - If they slow each other down then memory is probably a bottleneck.

Perturbing vector_add()

```
const int N = 50 * 1000 * 1000;

double A[N], B[N], C[N];

void vector_add()
{
    cilk_for (int i = 0; i < N; ++i) {
        A[i] = B[i] + C[i];
        fib(5);  // waste time
    }
}
```

Speedup of perturbed `vector_add()`



Interpreting the perturbed results

The memory is a bottleneck:

- A little extra work (`fib(5)`) keeps 8 cores busy. A little more extra work (`fib(10)`) keeps 16 cores busy.
- Thus, we have enough parallelism.
- The memory is *probably* a bottleneck. (If the machine had a shared FPU, the FPU could also be a bottleneck.)

OK, but how do you fix it?

- `vector_add` cannot be fixed in isolation.
- You must generally restructure your program to increase the reuse of cached data. Compare the iterative and recursive matrix multiplication from yesterday.
- (Or you can buy a newer CPU and faster memory.)

Lessons learned

- Memory is a common bottleneck.
- One way to diagnose bottlenecks is to perturb the program or the environment.
- Fixing memory bottlenecks usually requires algorithmic changes.

Example 4: Nested loops

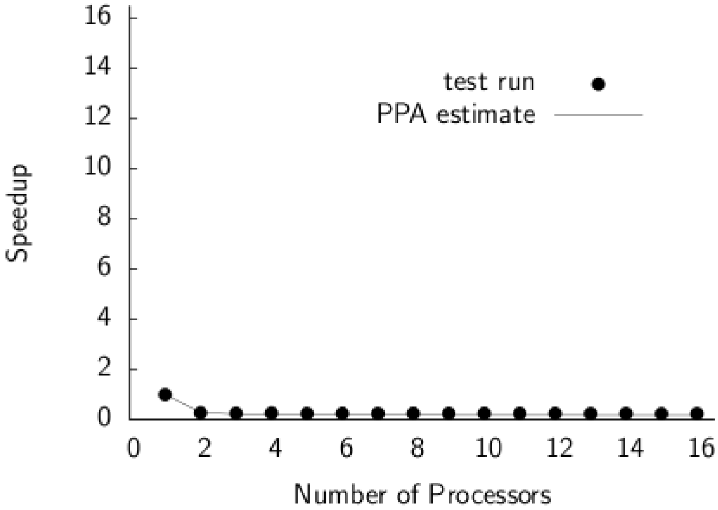
Code:

```
const int N = 1000 * 1000;

void inner_parallel()
{
    for (int i = 0; i < N; ++i)
        cilk_for (int j = 0; j < 4; ++j)
            fib(10); /* do some work */
}
```

Expectations:

- The inner loop does 4 things in parallel. The parallelism should be about 4.
- The PPA says that the parallelism is 3.6.
- We should see some speedup.

“Speedup” of `inner_parallel()`



Interchanging loops

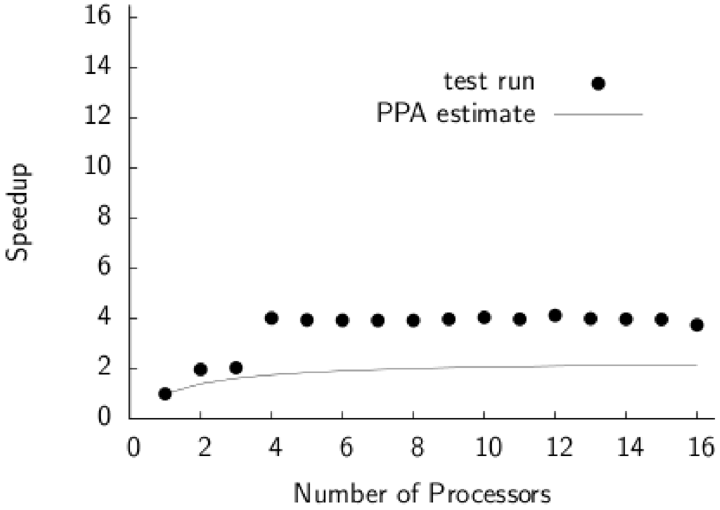
Code:

```
const int N = 1000 * 1000;

void outer_parallel()
{
    cilk_for (int j = 0; j < 4; ++j)
        for (int i = 0; i < N; ++i)
            fib(10); /* do some work */
}
```

Expectations:

- The outer loop does 4 things in parallel. The parallelism should be about 4.
- The PPA says that the parallelism is 4.
- Same as the previous program, which didn't work.

Speedup of `outer_parallel()`



Parallelism vs. burdened parallelism

Parallelism:

The best speedup you can hope for.

Burdened parallelism:

Parallelism after accounting for the unavoidable migration overheads.

Depends upon:

- How well we implement the Cilk++ scheduler.
- How you express the parallelism in your program.

The PPA prints the burdened parallelism:

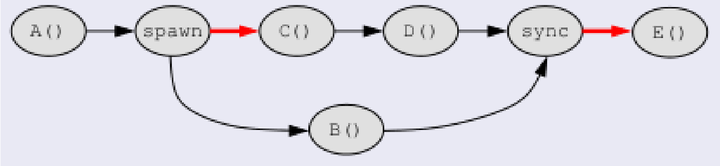
- 0.29 for `inner_parallel()`, 4.0 for `outer_parallel()`.
- In a good program, parallelism and burdened parallelism are about equal.

What is the burdened parallelism?

Code:

```
A();
cilk_spawn B();
C();
D();
cilk_sync;
E();
```

Burdened critical path:



The **burden** is $\Theta(10000)$ cycles (locks, malloc, cache warmup, reducers, etc.)

The burden in our examples

$\Theta(N)$ spawns/syncs on the critical path (large burden):

```
void inner_parallel()
{
    for (int i = 0; i < N; ++i)
        cilk_for (int j = 0; j < 4; ++j)
            fib(10); /* do some work */
}
```

$\Theta(1)$ spawns/syncs on the critical path (small burden):

```
void outer_parallel()
{
    cilk_for (int j = 0; j < 4; ++j)
        for (int i = 0; i < N; ++i)
            fib(10); /* do some work */
}
```

Lessons learned

- Insufficient parallelism yields *no speedup*; high burden yields *slowdown*.
- Many spawns but small parallelism: suspect large burden.
- The PPA helps by printing the burdened span and parallelism.
- The burden can be interpreted as the number of spawns/syncs on the critical path.
- If the burdened parallelism and the parallelism are approximately equal, your program is ok.

Sumary and notes

We have learned to identify and address these problems:

- High overhead due to small grain size in `cilk_for` loops.
- Insufficient parallelism.
- Insufficient memory bandwidth.
- Insufficient burdened parallelism.