Multithreaded Parallelism and Performance Measures

Marc Moreno Maza

University of Western Ontario, London, Ontario (Canada)

CS 4435 - CS 9624

Plan

1. Parallelism Complexity Measures
2. cilk_for Loops
3. Scheduling Theory and Implementation
4. Measuring Parallelism in Practice
5. Announcements

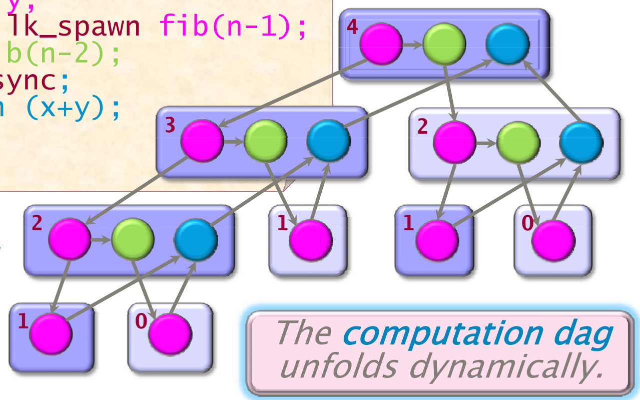
Plan

1. Parallelism Complexity Measures
2. cilk_for Loops
3. Scheduling Theory and Implementation
4. Measuring Parallelism in Practice
5. Announcements

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)`

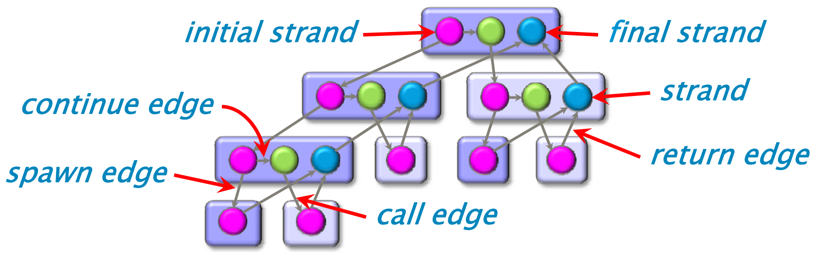


"Processor oblivious"

The computation dag unfolds dynamically.

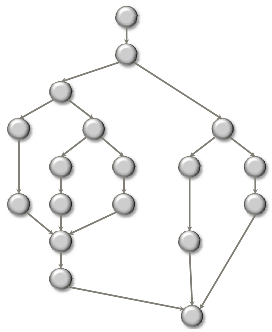
We shall also call this model **multithreaded parallelism**.

Terminology



- a **strand** is is a maximal sequence of instructions that ends with a `spawn`, `sync`, or `return` (either explicit or implicit) statement.
- At runtime, the *spawn* relation causes procedure instances to be structured as a rooted tree, called **spawn tree** or **parallel instruction stream**, where dependencies among strands form a dag.

Work and span



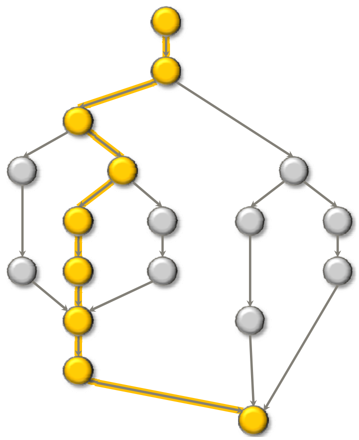
We define several performance measures. We assume an ideal situation: no cache issues, no interprocessor costs:

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.

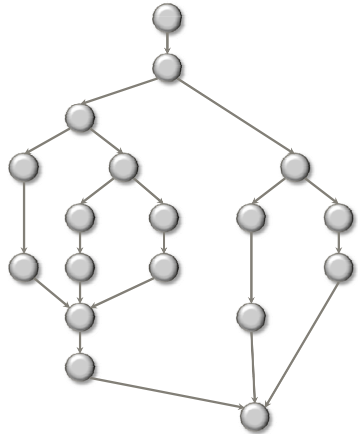
T_∞ is the minimum running time with infinitely many processors, called the **span**

The critical path length



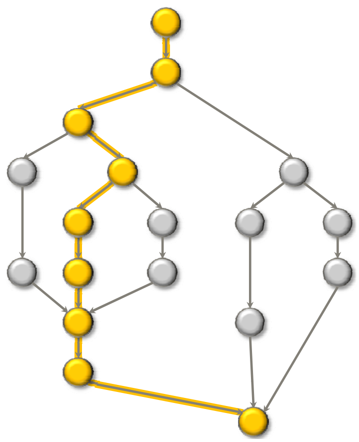
Assuming all strands run in unit time, the longest path in the DAG is equal to T_∞. For this reason, T_∞ is also referred to as the **critical path length**.

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_∞.

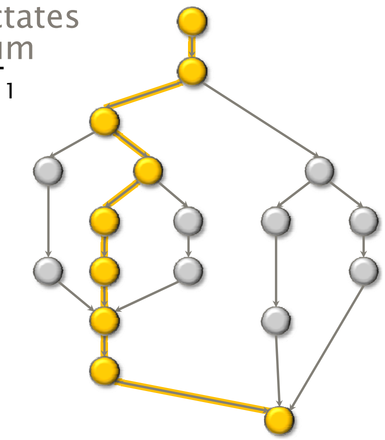
Speedup on p processors

- 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**: $T_1/T_P = o(p)$

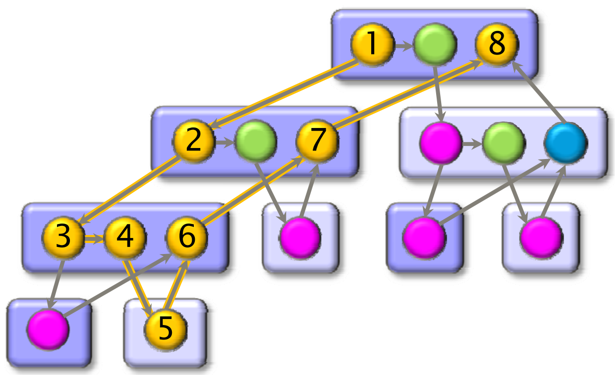
Parallelism

Because the Span Law dictates that $T_P \geq T_\infty$, the maximum possible speedup given T_1 and T_∞ is

T_1/T_∞ = *parallelism*
= the average amount of work per step along the span.



The Fibonacci example (1/2)

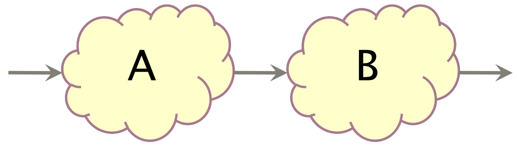


- For $\mathrm{Fib}(4)$, we have $T_1 = 17$ and $T_\infty = 8$ and thus $T_1/T_\infty = 2.125$.
- What about $T_1(\mathrm{Fib}(n))$ and $T_\infty(\mathrm{Fib}(n))$?

The Fibonacci example (2/2)

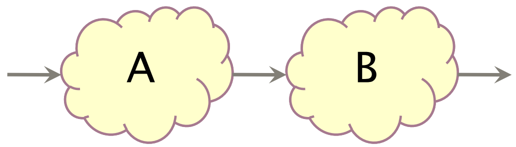
- We have $T_1(n) = T_1(n-1) + T_1(n-2) + \Theta(1)$. Let's solve it.
  - One verify by induction that $T(n) \leq aF_n - b$ for $b > 0$ large enough to dominate $\Theta(1)$ and $a > 1$.
  - We can then choose a large enough to satisfy the initial condition, whatever that is.
  - On the other hand we also have $F_n \leq T(n)$.
  - Therefore $T_1(n) = \Theta(F_n) = \Theta(\psi^n)$ with $\psi = (1 + \sqrt{5})/2$.
- We have $T_\infty(n) = \max(T_\infty(n-1), T_\infty(n-2)) + \Theta(1)$.
  - We easily check $T_\infty(n-1) \geq T_\infty(n-2)$.
  - This implies $T_\infty(n) = T_\infty(n-1) + \Theta(1)$.
  - Therefore $T_\infty(n) = \Theta(n)$.
- Consequently the parallelism is $\Theta(\psi^n/n)$.

Series composition



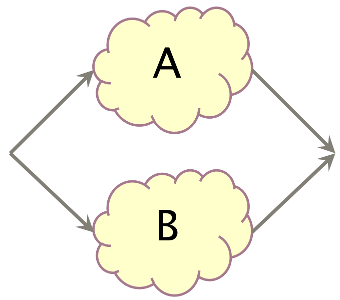
- Work?
- Span?

Series composition



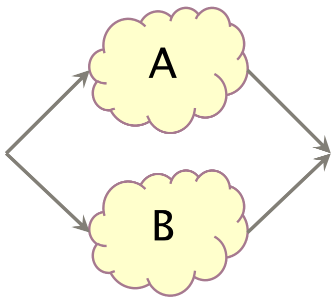
- Work: $T_1(A \cup B) = T_1(A) + T_1(B)$
- Span: $T_\infty(A \cup B) = T_\infty(A) + T_\infty(B)$

Parallel composition



- Work?
- Span?

Parallel composition



- Work: $T_1(A \cup B) = T_1(A) + T_1(B)$
- Span: $T_\infty(A \cup B) = \max(T_\infty(A), T_\infty(B))$

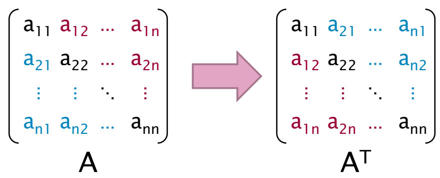
Some results in the fork-join parallelism model

Algorithm	Work	Span
Merge sort	$\Theta(n \lg n)$	$\Theta(\lg^3 n)$
Matrix multiplication	$\Theta(n^3)$	$\Theta(\lg n)$
Strassen	$\Theta(n^{\lg 7})$	$\Theta(\lg^2 n)$
LU-decomposition	$\Theta(n^3)$	$\Theta(n \lg n)$
Tableau construction	$\Theta(n^2)$	$\Omega(n^{\lg 3})$
FFT	$\Theta(n \lg n)$	$\Theta(\lg^2 n)$
Breadth-first search	$\Theta(E)$	$\Theta(d \lg V)$

We shall prove those results in the next lectures.

Plan

For loop parallelism in Cilk++




```
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;
    }
}
```

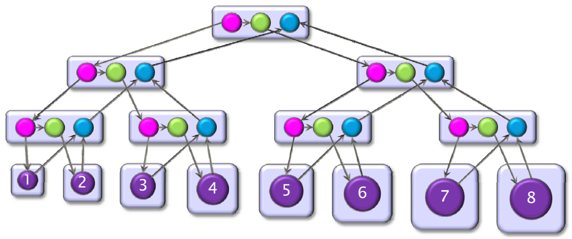
The iterations of a cilk_for loop execute in parallel.

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**: $\Theta(n)$

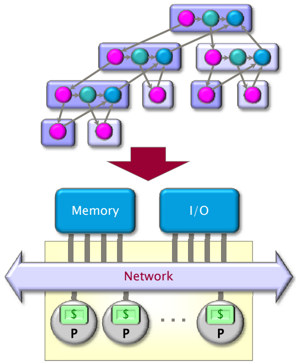
Parallelizing the inner loop

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

- **Span of outer loop control**: $\Theta(\log(n))$
- **Max span of an inner loop control**: $\Theta(\log(n))$
- **Span of an iteration**: $\Theta(1)$
- **Span**: $\Theta(\log(n))$
- **Work**: $\Theta(n^2)$
- **Parallelism**: $\Theta(n^2/\log(n))$ **But! More on this next week ...**

Plan

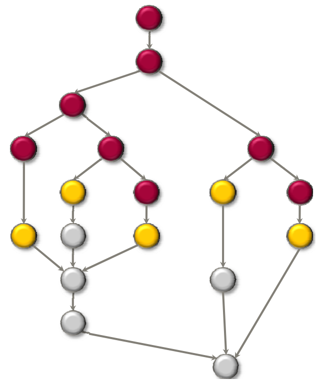
Scheduling



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

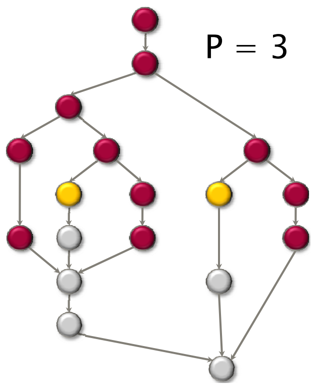
- If decisions are made at runtime, the scheduler is *online*, otherwise, it is *offline*
- `Cilk++`'s scheduler maps strands onto processors dynamically at runtime.

Greedy scheduling (1/2)



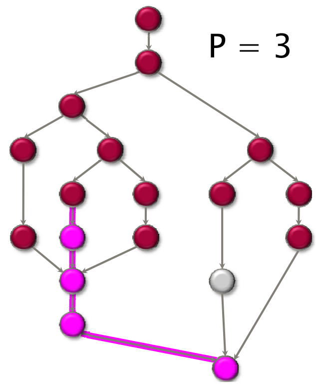
- A strand is **ready** if all its predecessors have executed
- A scheduler is **greedy** if it attempts to do as much work as possible at every step.

Greedy scheduling (2/2)



- In any *greedy schedule*, there are two types of steps:
  - **complete step**: There are at least p strands that are ready to run. The greedy scheduler selects any p of them and runs them.
  - **incomplete step**: There are strictly less than p threads that are ready to run. The greedy scheduler runs them all.

Theorem of Graham and Brent



For any greedy schedule, we have $T_p \leq T_1/p + T_\infty$

- #complete steps $\leq T_1/p$, by definition of T_1.
- #incomplete steps $\leq T_\infty$. Indeed, let G' be the subgraph of G that remains to be executed immediately prior to a incomplete step.
  - (*i*) During this incomplete step, all strands that can be run are actually run
  - (*ii*) Hence removing this incomplete step from G' reduces T_∞ by one.

Corollary 1

A greedy scheduler is always within a factor of 2 of optimal.

From the work and span laws, we have:

$$T_P \geq \max(T_1/p, T_\infty) \tag{1}$$

In addition, we can trivially express:

$$T_1/p \leq \max(T_1/p, T_\infty) \tag{2}$$

$$T_\infty \leq \max(T_1/p, T_\infty) \tag{3}$$

From Graham - Brent Theorem, we deduce:

$$\begin{aligned} T_P &\leq T_1/p + T_\infty & (4)\\ &\leq \max(T_1/p, T_\infty) + \max(T_1/p, T_\infty) & (5)\\ &\leq 2\max(T_1/p, T_\infty) & (6) \end{aligned}$$

which concludes the proof.

Corollary 2

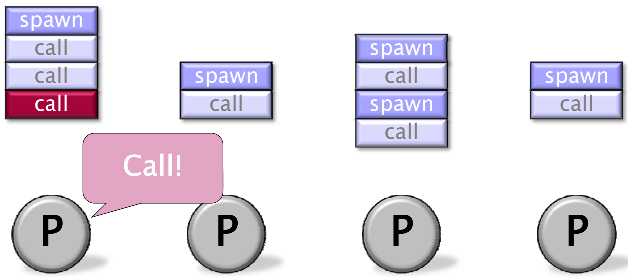
The greedy scheduler achieves linear speedup whenever $T_\infty = O(T_1/p)$.

From Graham - Brent Theorem, we deduce:

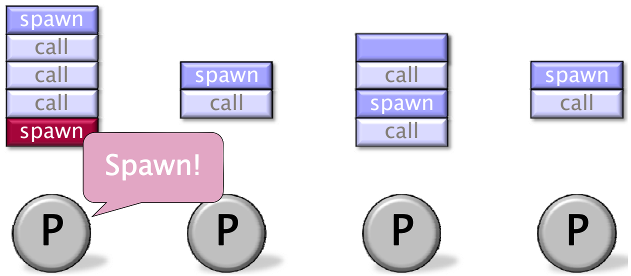
$$\begin{aligned} T_p &\leq T_1/p + T_\infty & (7)\\ &= T_1/p + O(T_1/p) & (8)\\ &= \Theta(T_1/p) & (9) \end{aligned}$$

The idea is to operate in the range where T_1/p dominates T_∞. As long as T_1/p dominates T_∞, all processors can be used efficiently.
The quantity T_1/pT_∞ is called the **parallel slackness**.

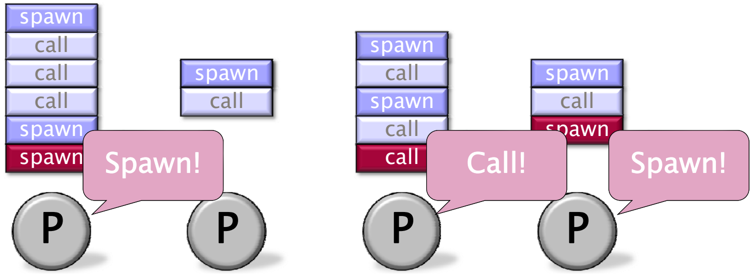
The work-stealing scheduler (1/11)



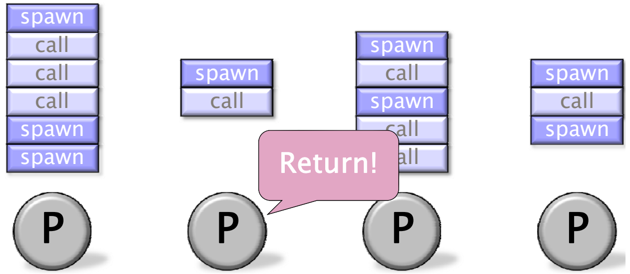
The work-stealing scheduler (2/11)



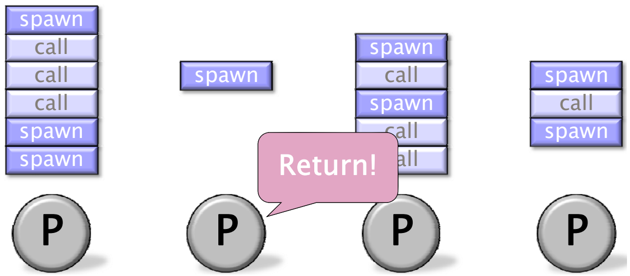
The work-stealing scheduler (3/11)



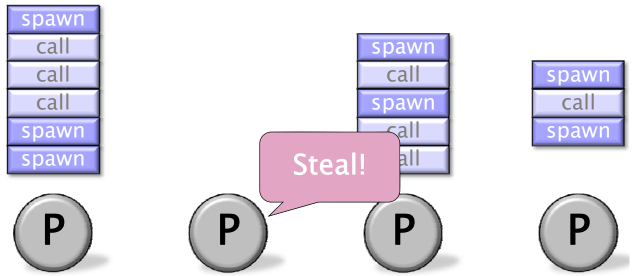
The work-stealing scheduler (4/11)



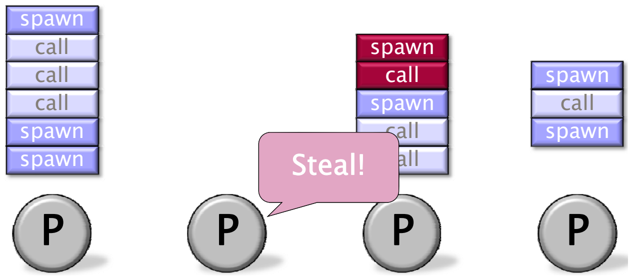
The work-stealing scheduler (5/11)



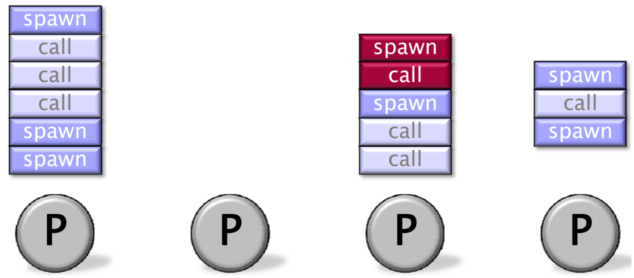
The work-stealing scheduler (6/11)



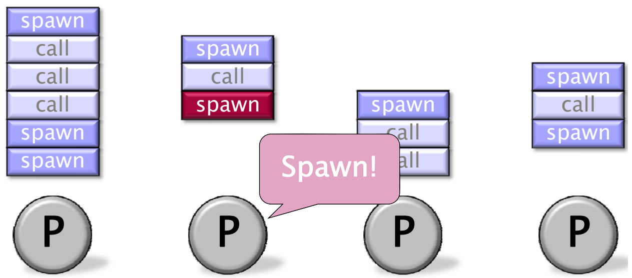
The work-stealing scheduler (7/11)



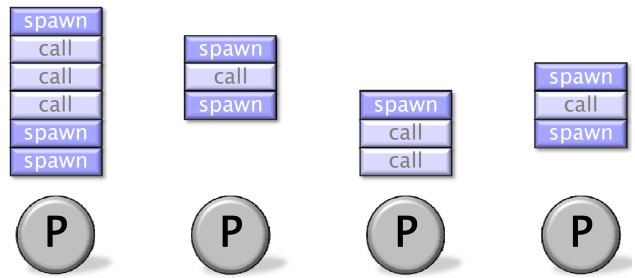
The work-stealing scheduler (8/11)



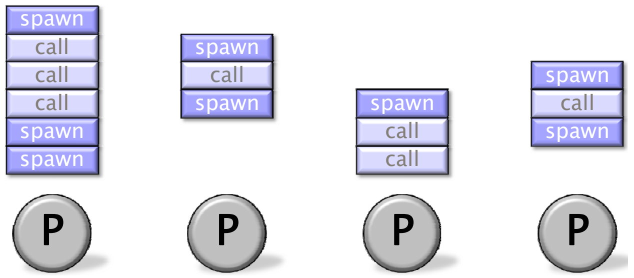
The work-stealing scheduler (9/11)



The work-stealing scheduler (10/11)



The work-stealing scheduler (11/11)



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

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

- During a steal-free parallel steps (steps at which all processors have work on their deque) each of the p processors consumes 1 work unit.
- Thus, there is at most T_1/p steal-free parallel steps.
- During a parallel step with steals each thief may reduce by 1 the running time with a probability of $1/p$
- Thus, the expected number of steals is $O(p\,T_\infty)$.
- Therefore, the expected running time

$$T_P = (T_1 + O(p\,T_\infty))/p = T_1/p + O(T_\infty). \quad (10)$$

Overheads and burden

- Obviously $T_1/p + T_\infty$ will over-estimate T_p in practice.
- 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.
- 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

Span overhead

- Let T_1, T_∞, T_p be given. We want to refine the *randomized work-stealing complexity result.*
- The **span overhead** is the smallest constant c_∞ such that

$$T_p \leq T_1/p + c_\infty T_\infty.$$

- Recall that T_1/T_∞ is the maximum possible speed-up that the application can obtain.
- We call **parallel slackness assumption** the following property

$$T_1/T_\infty >> c_\infty p \quad (11)$$

  that is, $c_\infty\, p$ is much smaller than the average parallelism .
- Under this assumption it follows that $T_1/p >> c_\infty T_\infty$ holds, thus c_∞ has little effect on performance when sufficiently slackness exists.

Work overhead

- Let T_s be the running time of the C++ elision of a `Cilk++` program.
- We denote by c_1 the **work overhead**

$$c_1 = T_1/T_s$$

- Recall the expected running time: $T_P \leq T_1/P + c_\infty T_\infty$. Thus with the parallel slackness assumption we get

$$T_P \leq c_1 T_s/p + c_\infty T_\infty \simeq c_1 T_s/p. \quad (12)$$

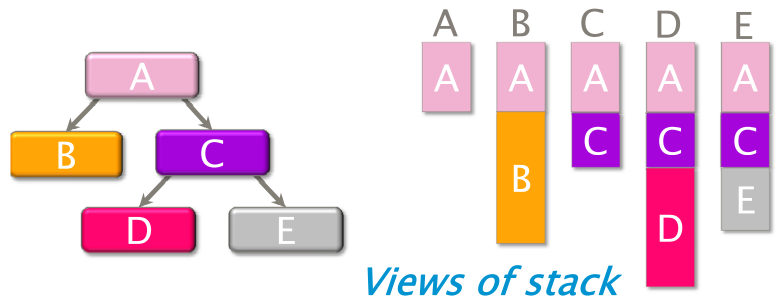
- We can now state the **work first principle** precisely

  Minimize c_1 , even at the expense of a larger c_∞.

  This is a key feature since it is conceptually easier to minimize c_1 rather than minimizing c_∞.

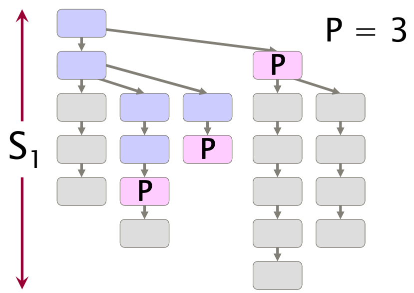
- `Cilk++` estimates T_p as $T_p = T_1/p + 1.7\ \mathrm{burden\_span}$, where $\mathrm{burden\_span}$ is 15000 instructions times the number of continuation edges along the critical path.

The cactus stack



- A **cactus stack** is used to implement C's rule for sharing of function-local variables.
- A stack frame can only see data stored in the current and in the previous stack frames.

Space bounds



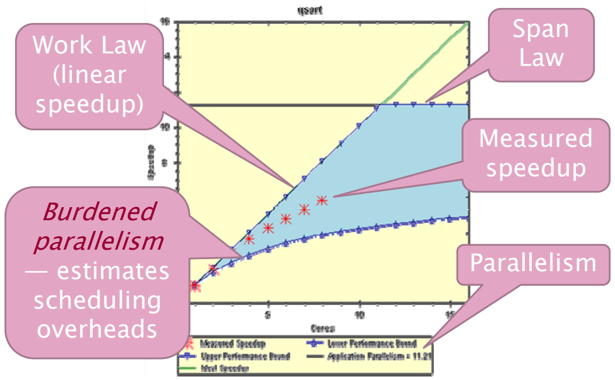
The space S_p of a parallel execution on p processors required by `Cilk++`'s work-stealing satisfies:

$$S_p \leq p \cdot S_1 \quad (13)$$

where S_1 is the minimal serial space requirement.

Plan

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

Plan

Acknowledgements

- Charles E. Leiserson (MIT) for providing me with the sources of its lecture notes.
- Matteo Frigo (Intel) for supporting the work of my team with Cilk++.
- Yuzhen Xie (UWO) for helping me with the images used in these slides.
- Liyun Li (UWO) for generating the experimental data.

References

- Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The Implementation of the Cilk-5 Multithreaded Language. Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation, Pages: 212-223. June, 1998.
- Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An Efficient Multithreaded Runtime System. Journal of Parallel and Distributed Computing, 55-69, August 25, 1996.
- Robert D. Blumofe and Charles E. Leiserson. Scheduling Multithreaded Computations by Work Stealing. Journal of the ACM, Vol. 46, No. 5, pp. 720-748. September 1999.