Cilk and OpenMP (Lecture 20, cs262a)

Ion Stoica,
UC Berkeley
April 4, 2018

Today's papers

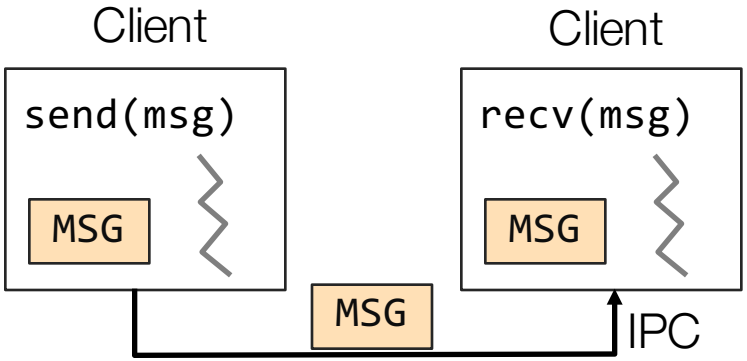
"Cilk: An Efficient Multithreaded Runtime System" by Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall and Yuli Zhou

http://supertech.csail.mit.edu/papers/PPoPP95.pdf

"OpenMP: An IndustryStandard API for SharedMemory Programming", Leonardo Dagum and Ramesh Menon

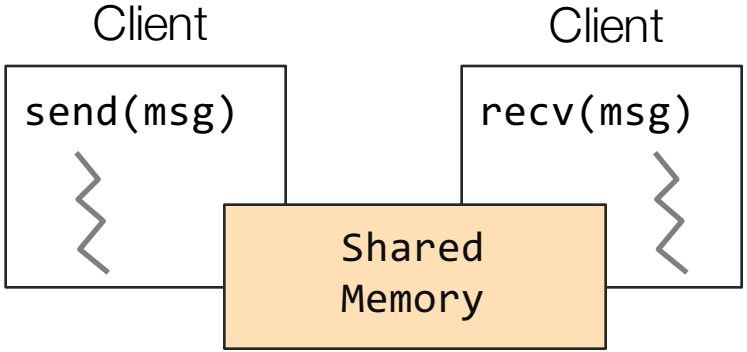
https://ucbrise.github.io/cs262a-spring2018/

Message passing vs. Shared memory



Message passing: exchange data explicitly via IPC

Application developers define protocol and exchanging format, number of participants, and each exchange



Shared memory: all multiple processes to share data via memory

Applications must locate and and map shared memory regions to exchange data

Architectures

Uniformed Shared Memory (UMA)
Cray 2

Non-Uniformed Shared Memory (NUMA)
SGI Altix 3700

Massively Parallel
DistrBluegene/L

Orthogonal to programming model

Motivation

Multicore CPUs are everywhere:

- Servers with over 100 cores today
- Even smartphone CPUs have 8 cores

Multithreading, natural programming model

- All processors share the same memory
- Threads in a process see same address space
- Many shared-memory algorithms developed

But…

Multithreading is hard

- Lots of expertise necessary
- Deadlocks and race conditions
- Non-deterministic behavior makes it hard to debug

Example

Parallelize the following code using threads:

```
for (i=0; i<n; i++) {
  sum = sum + sqrt(sin(data[i]));
}
```

Why hard?

- Need mutex to protect the accesses to sum
- Different code for serial and parallel version
- No built-in tuning (# of processors?)

Cilk

Based on slides available at http://supertech.csail.mit.edu/cilk/lecture-1.pdf

Cilk

A C language for programming dynamic multithreaded applications on shared-memory multiprocessors

Examples:

- dense and sparse matrix computations
- n-body simulation
- heuristic search
- graphics rendering
- …

Cilk in one slide

- Simple extension to C; just **three** basic keywords
- Cilk programs maintain serial semantics
  - Abstracts away parallel execution, load balancing and scheduling
- Parallelism
  - Processor-oblivious
  - Speculative execution
- Provides performance “guarantees”

Example: Fibonacci

C (elision)

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

Click

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

- A Cilk program's serial elision is always a legal implementation of Cilk semantics
- Cilk provides no new data types.

Cilk basic kewords

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

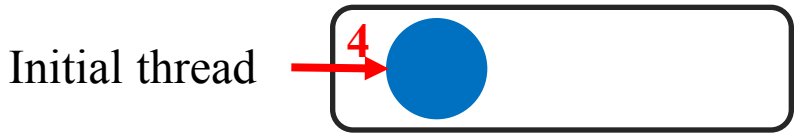
Identifies a function as a Cilk **procedure**, capable of being spawned in parallel.

The named child Cilk procedure can execute in parallel with the **parent caller**

Control cannot pass this point until all spawned children have returned

Dynamic multithreading – example: fib(4)

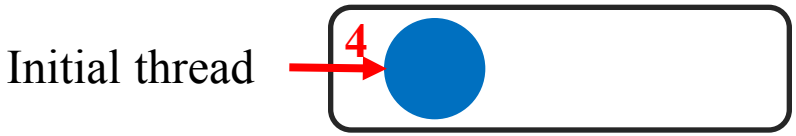
```
click int fib (int n) {
  if (n<2) return (n);
  else {
    int x,y;
    x = spawn fib(n-1);
    y = spawn fib(n-2);
    sync
    return (x+y);
  }
}
```



Computation unfolds **dynamically**

Dynamic multithreading – example: fib(4)

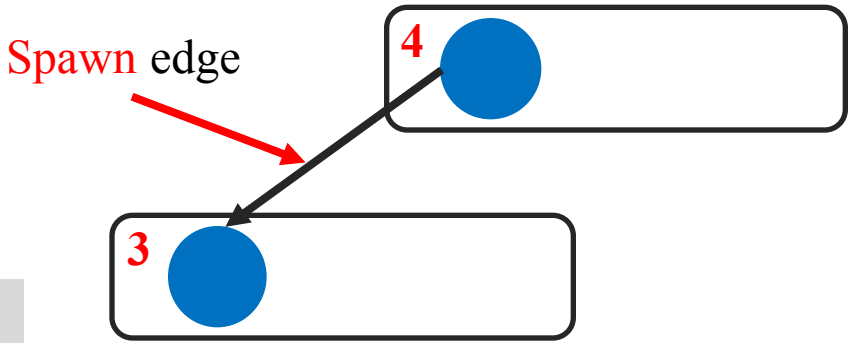
```
click int fib (int n) {
  if (n<2) return (n);
  else {
    int x,y;
    x = spawn fib(n-1);
    y = spawn fib(n-2);
    sync
    return (x+y);
  }
}
```



- Execution represented as a graph, G = (V, E)
- Each vertex v ∈ V represents a (Cilk) thread: a maximal sequence of instructions not containing parallel control (spawn, sync, return)
- Every edge e ∈ E is either a spawn, a return or a continue edge

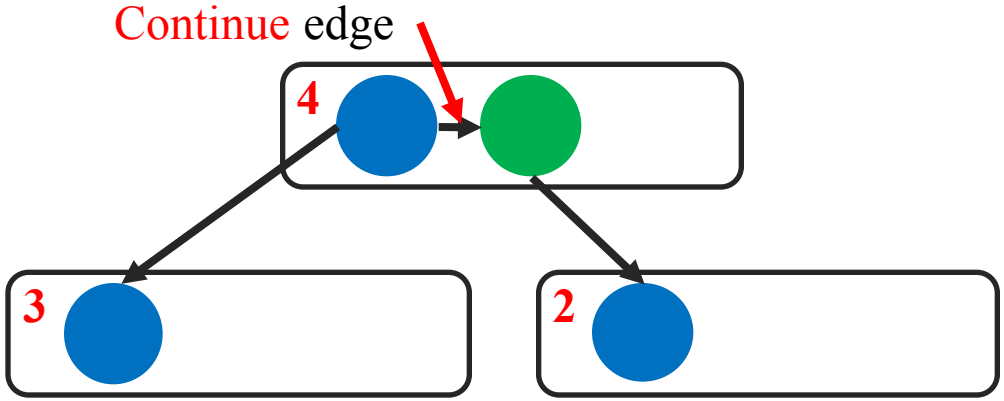
Dynamic multithreading – example: fib(4)

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



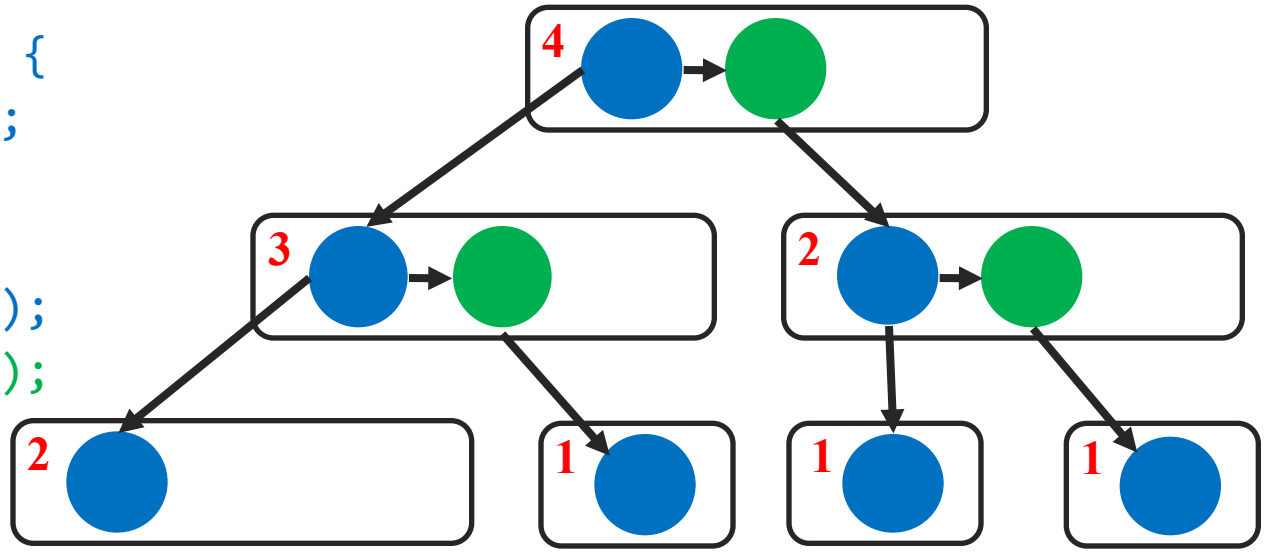
Dynamic multithreading – example: fib(4)

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



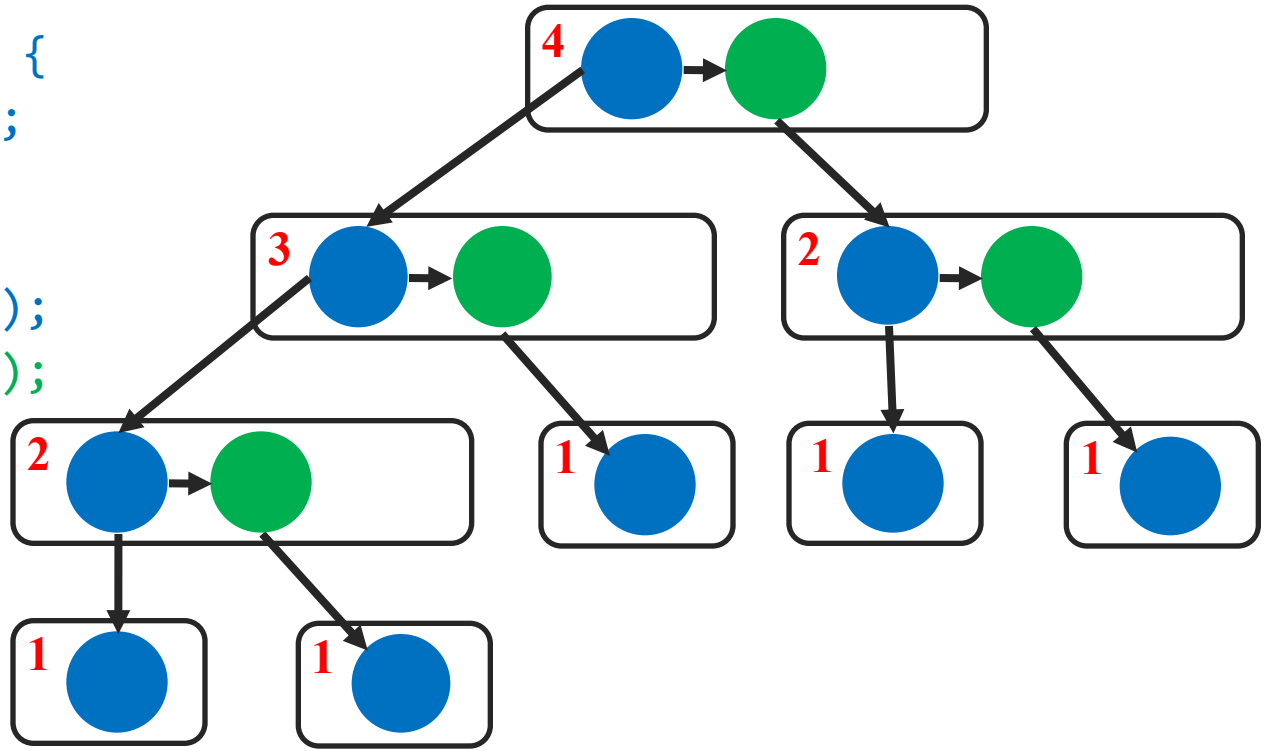
Dynamic multithreading – example: fib(4)

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



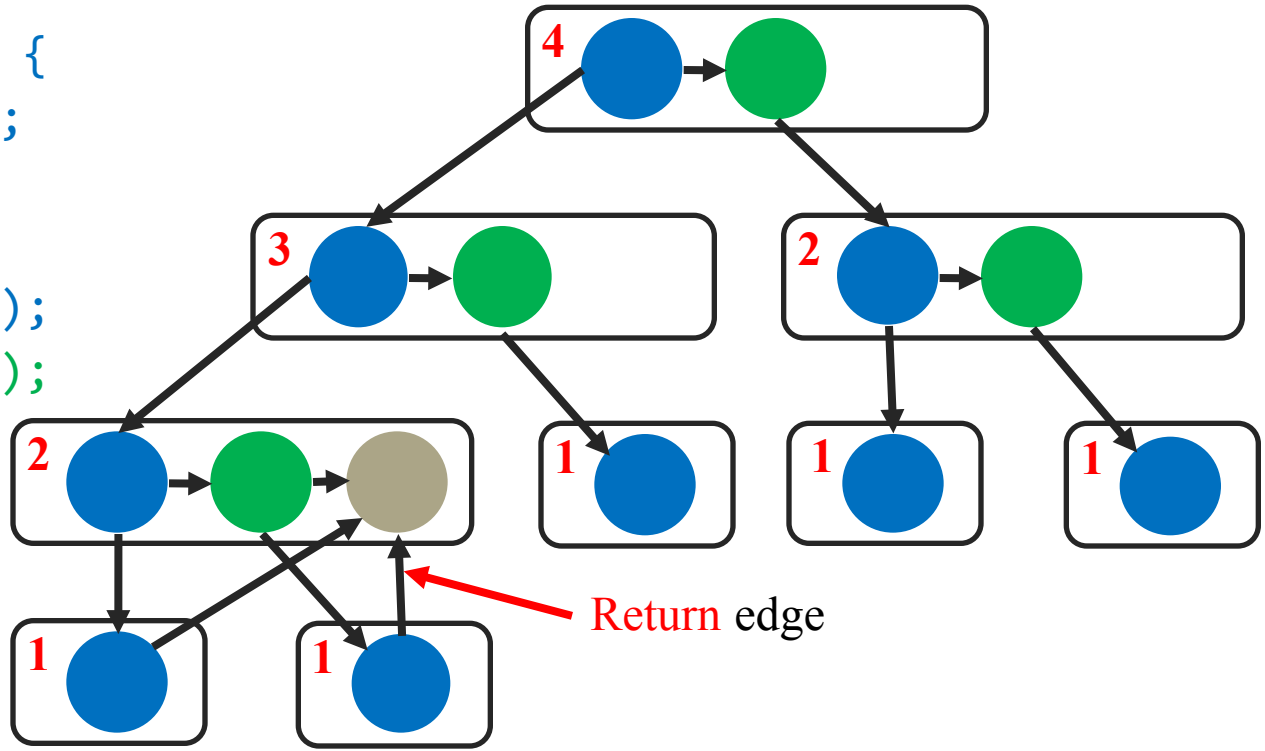
Dynamic multithreading – example: fib(4)

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



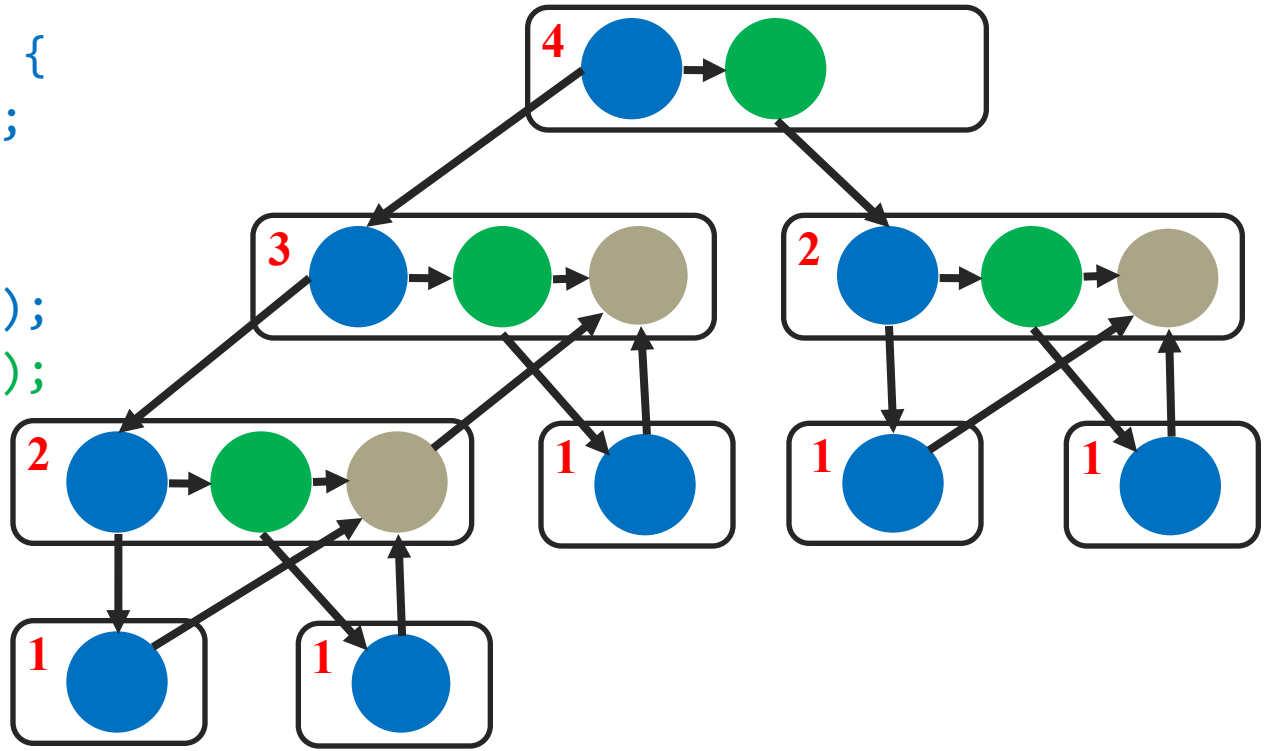
Dynamic multithreading – example: fib(4)

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



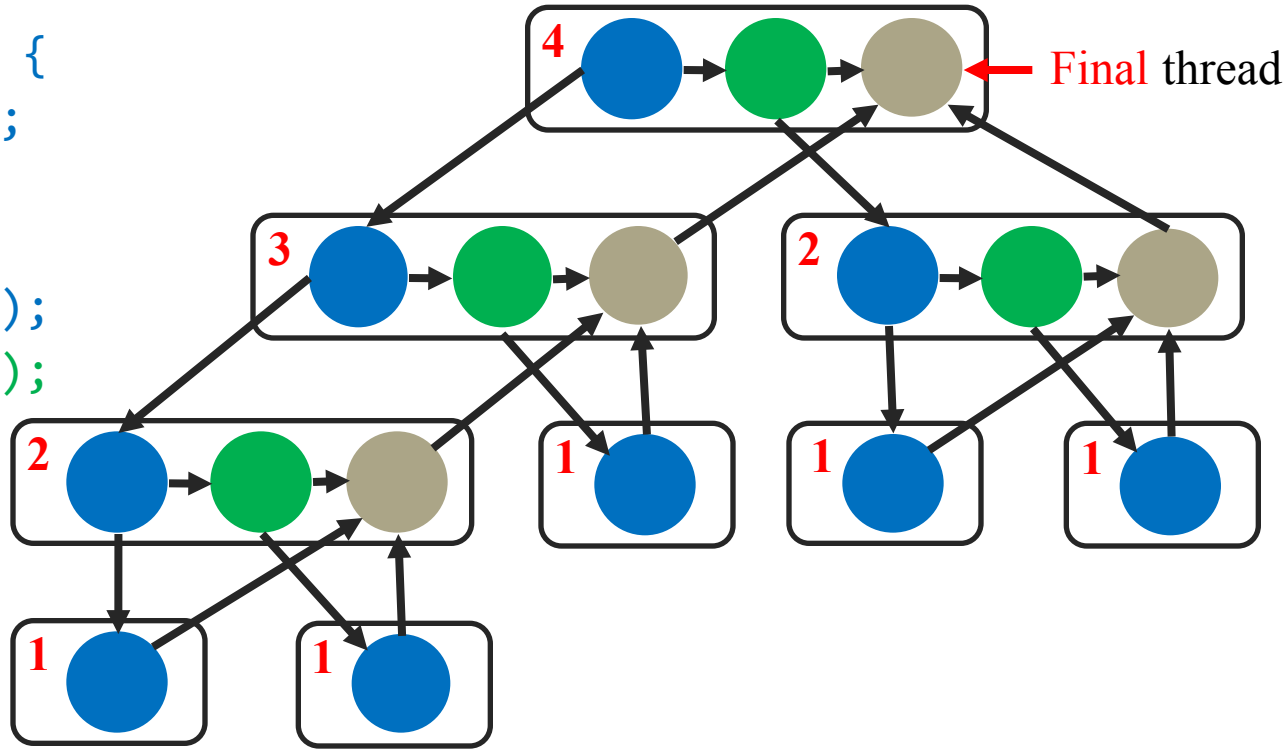
Dynamic multithreading – example: fib(4)

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



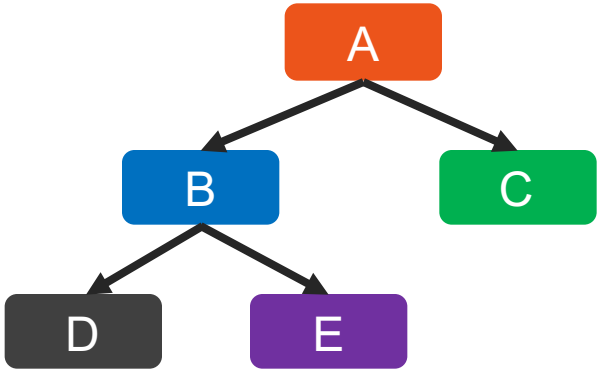
Dynamic multithreading – example: fib(4)

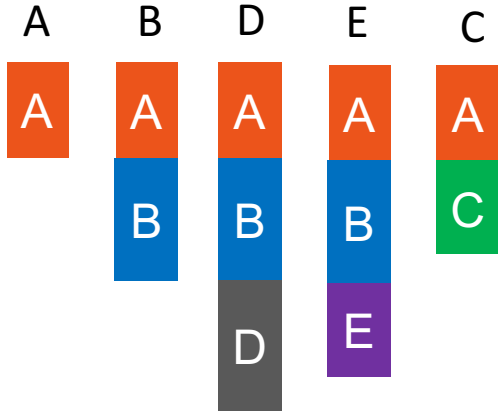
```
click int fib (int n) {
  if (n<2) return (n);
  else {
    int x,y;
    x = spawn fib(n-1);
    y = spawn fib(n-2);
    sync
    return (x+y);
  }
}
```



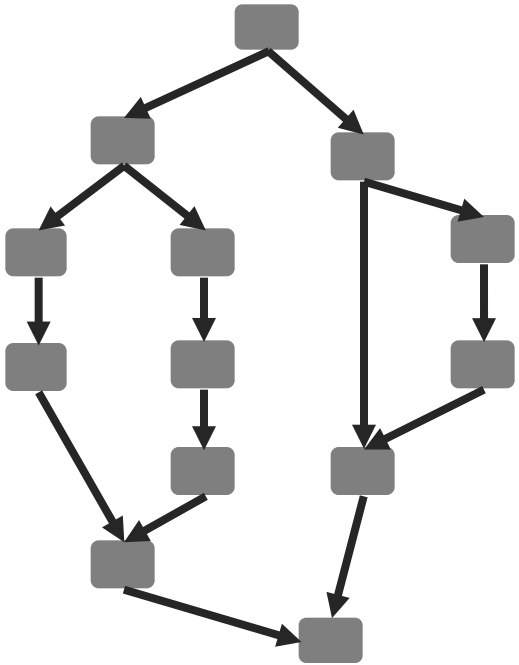
Cactus stack

- A stack pointer can be passed from parent to child, but not from child to parent
  - Support several views of stack
- Cilk also supports malloc



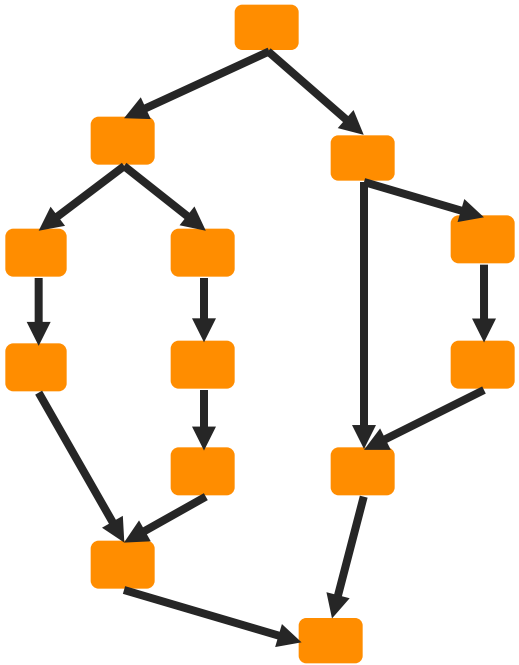


Algorithmic complexity



T_P = execution time on P processors

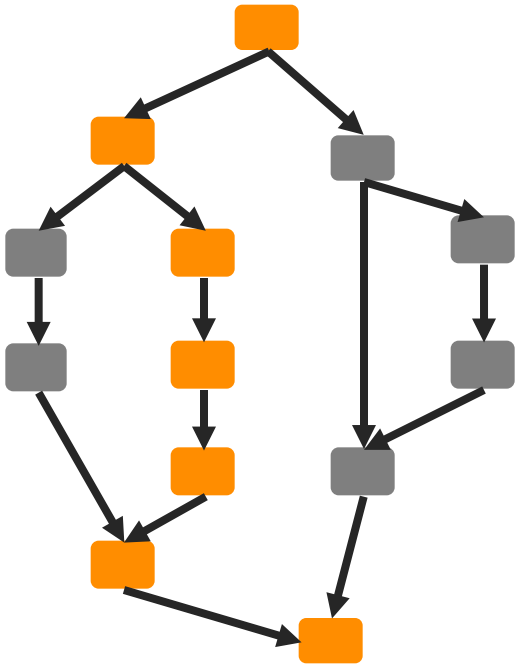
Algorithmic complexity



T_P = execution time on P processors

T_1 = total work

Algorithmic complexity

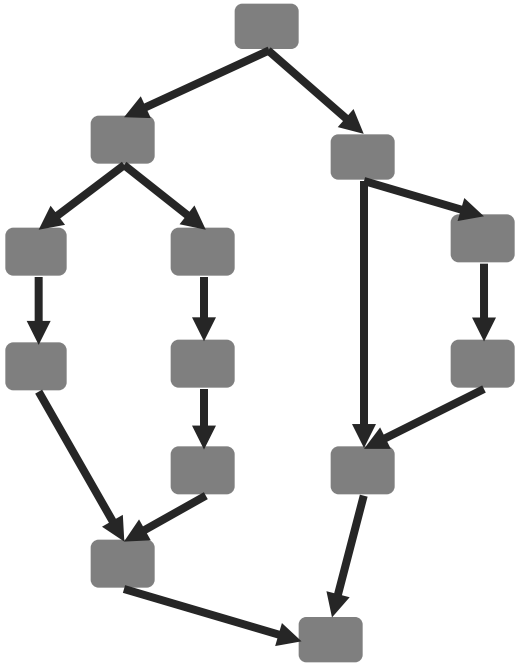


T_P = execution time on P processors

T_1 = total work

T_C = critical path length (span)

Algorithmic complexity



T_P = execution time on P processors

T_1 = total work

T_C = critical path length (span)

Lower bounds

$T_P >= T_1/P$

$T_P >= T_C$

Speedup

T_1/T_P = speedup on P processors.

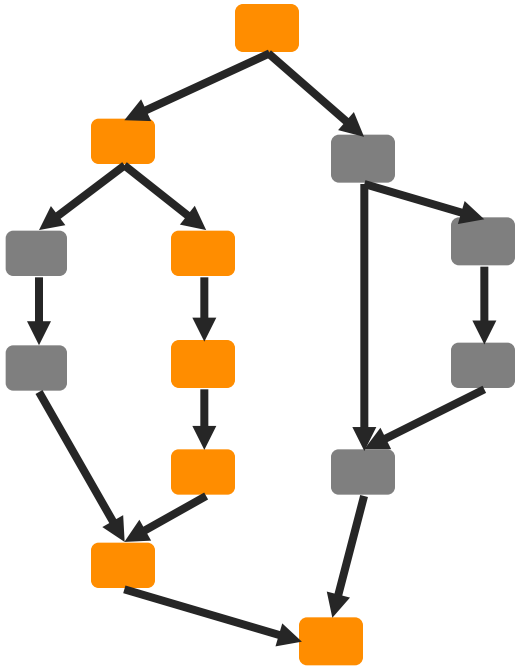
If $T_1/T_P = \Theta(P) <= P$, linear speedup

If $T_1/T_P = P$, perfect linear speedup

Parallelism

Since $T_P >= T_C$, T_1/T_C is maximum speedup

T_1/T_P = prallelism, average amount of work per step along the span

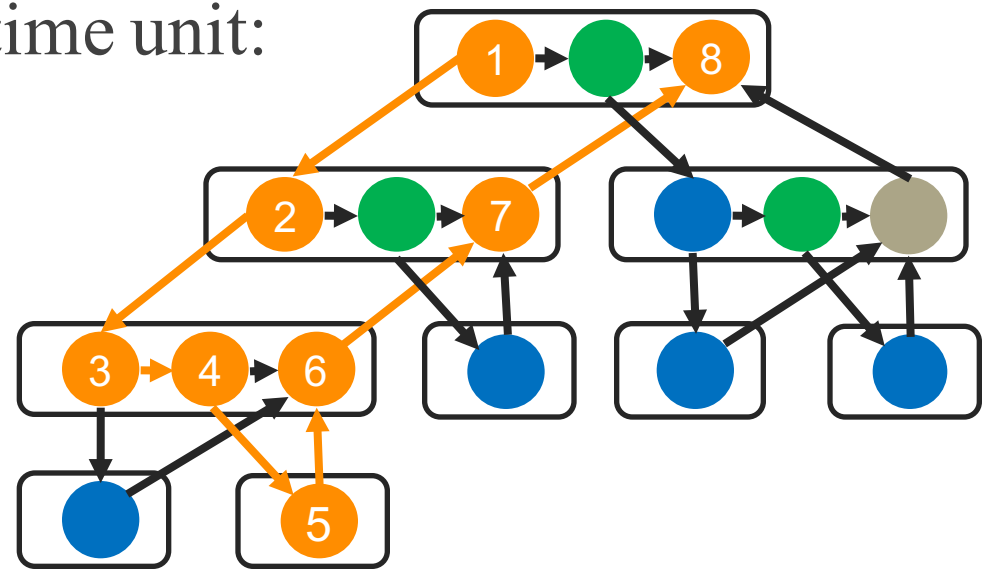


Example: fib(4)

Assume each thread takes 1 time unit:

- Work: $T_1 = 17$
- Span: $T_C = 8$
- Parallelism: $T_1/T_C = 17/8$

Little sense to use more than 2 processors!



Example: vector addition

```
void vadd (real *A, real *B, int n){
  int i; for (i=0; i<n; i++) A[i]+=B[i];
}
```

Example: vector addition

```
void vadd (real *A, real *B, int n){
  int i; for (i=0; i<n; i++) A[i]+=B[i];
}
```

Key idea: convert loops to recursion..

Example: vector addition

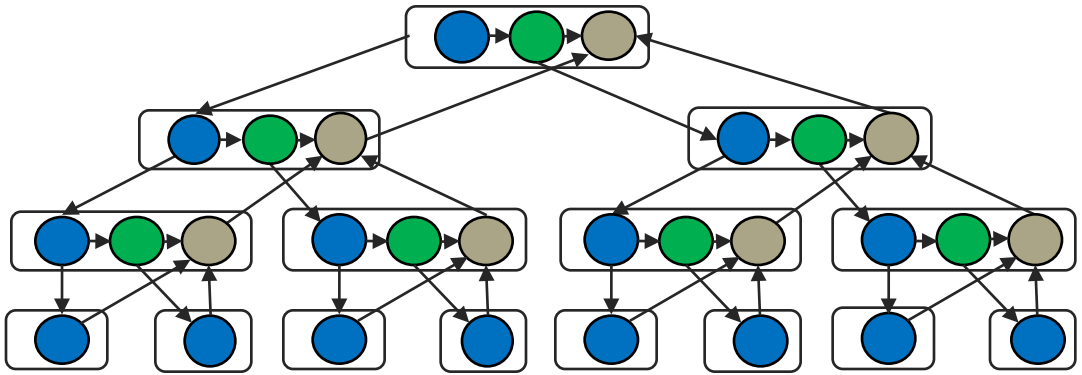
```
void vadd (real *A, real *B, int n){
  int i; for (i=0; i<n; i++) A[i]+=B[i];
}
```

Key idea: convert loops to recursion..

```
void vadd (real *A, real *B, int n) {
  if (n<=BASE) {
    int i; for (i=0; i<n; i++) A[i]+=B[i];
  } else {
    vadd (A, B, n/2);
    vadd (A+n/2, B+n/2, n-n/2);
  }
}
```

Example: vector addition

```
cilk void vadd (real *A, real *B, int n) {
  if (n<=BASE) {
    int i; for (i=0; i<n; i++) A[i]+=B[i];
  } else {
    spawn vadd (A, B, n/2);
    spawn vadd (A+n/2, B+n/2, n-n/2);
    sync;
  }
}
```



BASE

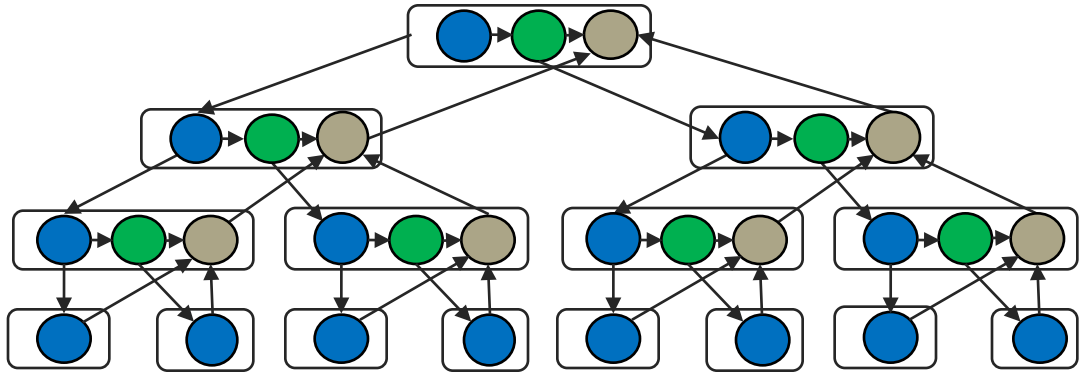
Example: vector addition

Assume BASE = $\Theta(1)$:

Work: $T_1 = \Theta(n)$

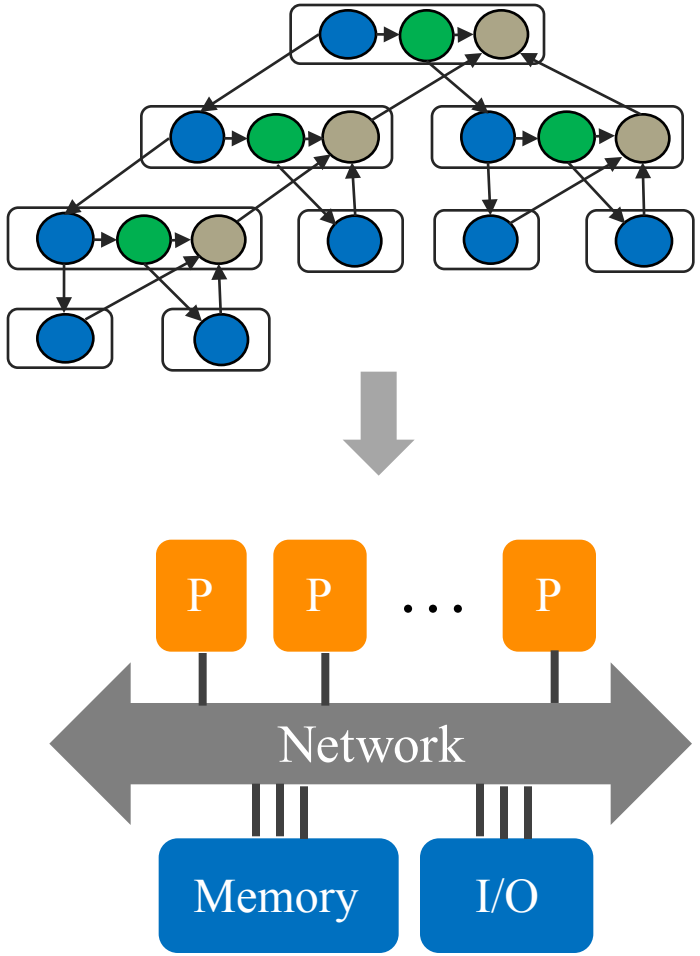
Span: $T_C = \Theta(\log n)$

Parallelism: $T_1/T_C = \Theta(n/\log n)$



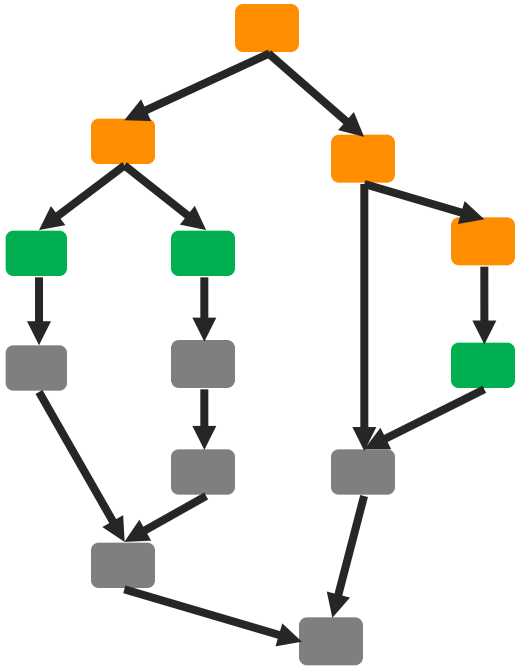
Scheduling

- Cilk scheduler maps Cilk threads onto processors dynamically at runtime



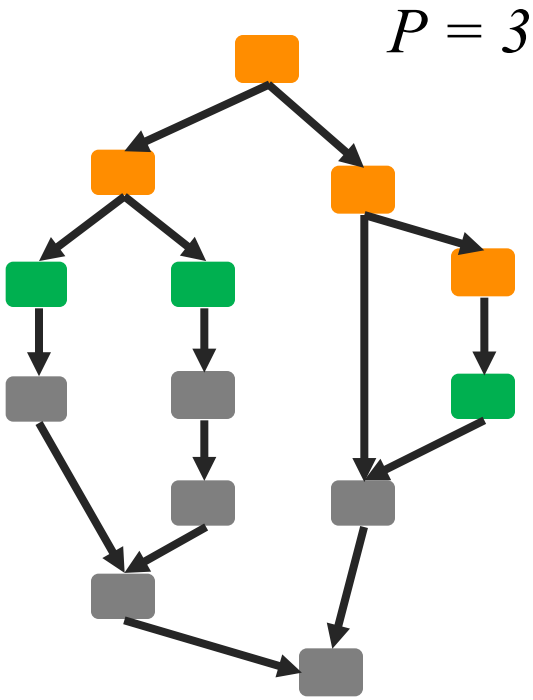
Greedy scheduling

- Key idea: Do as much as possible on every step
- Definition: A thread is ready if all its predecessors have executed



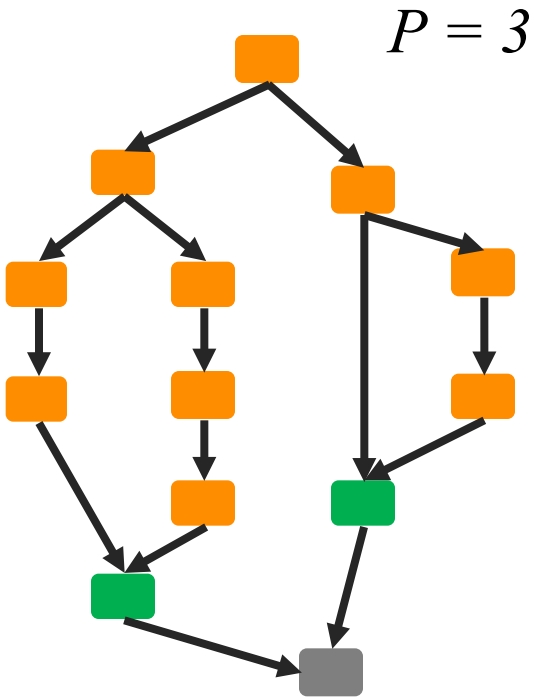
Greedy sechduling

- Key idea: Do as much as possible on every step
- Definition: A thread is ready if all its predecessors have executed
- Complete step
  - If # ready threads >= P, run any P ready threads



Greedy scheduling

- Key idea: Do as much as possible on every step
- Definition: A thread is ready if all its predecessors have executed
- Complete step
  - If # ready threads >= P, run any P ready threads
- Incomplete step
  - If # ready threads < P run all P ready threads



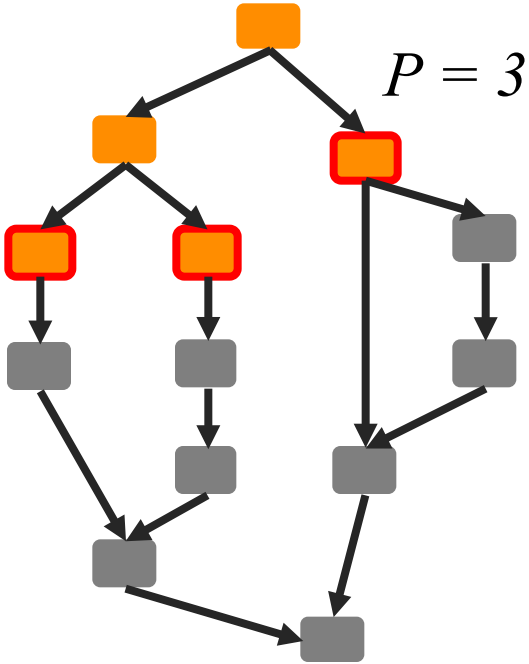
Greedy-Scheduling Theorem

Theorem [Graham '68 & Brent '75] Any greedy scheduler achieves

$$T_P \leq T_1/P + T_C$$

Proof sketch:

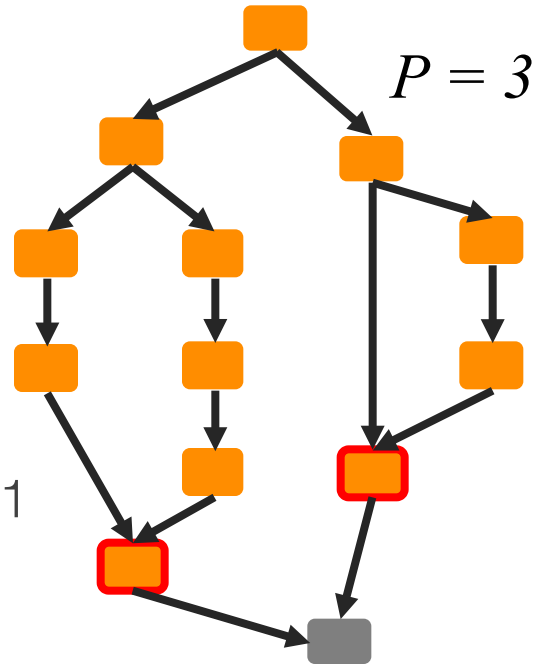
- # complete steps $\leq T_1/P$ since each complete step performs P work



Greedy-Scheduling Theorem

Theorem [Graham '68 & Brent '75] Any greedy scheduler achieves

$$T_P \leq T_1/P + T_C$$



Proof sketch:

- # complete steps $\leq T_1/P$ since each complete step performs P work
- # incomplete steps $\leq T_C$, since each incomplete step reduces the span of the unexecuted DAG by 1

Optimality of greedy

Corollary: Any greedy scheduler is within a factor of 2 of optimal

Proof: Let T_P^* be execution time produced by optimal scheduler

Since $T_P^* \geq \max\{T_1/P, T_\infty\}$ (lower bounds), we have

$$\begin{aligned} T_P &\leq T_1/P + T_\infty \\ &\leq 2 \max\{T_1/P, T_\infty\} \\ &\leq 2\, T_P^* \end{aligned}$$

Linear speedup

Corollary: Any greedy scheduler achieves near-perfect linear speedup whenever $P << T_1/T_C$.

Proof: From $P << T_1/T_C$ we get $T_C << T_1/P$

From the Greedy Scheduling Theorem gives us

$$T_P \leq T_1/P + T_C \approx T_1/P$$

Thus, speedup is $T_1/T_P \approx P$

Work stealing

Each processor has a queue of threads to run

A spawned thread is put on *local* processor queue

When a processor runs out of work, it looks at queues of other processors and "steals" their work items

- Typically pick the processor from where to steal randomly

Cilk performance

Cilk's "work-stealing" scheduler achieves

$$T_P = T_1/P + \mathrm{O}(T_C)$$ expected time (provably);

$$T_P \approx T_1/P + \mathrm{O}(T_C)$$ time (empirically).

Near-perfect linear speedup if $P << T_1/T_C$

- Instrumentation in Cilk allows to accurately measure T_1 and T_C
- The average cost of a spawn in Cilk-5 is only 2–6 times the cost of an ordinary C function call, depending on the platform

Summary

C extension for multithreaded programs

- Now available also for C++: Cilk Plus from Intel

Simple; only three keywords: cilk, spawn, sync

- Cilk Plus has actually only two: cilk_spaw, cilk_sync
- Equivalent to serial program on a single core
- Abstracts away parallelism, load balancing and scheduling

Leverages recursion pattern

- Might need to rewrite programs to fit the pattern (see vector addition example)

OpenMP

Based on the “Introduction to OpenMP” presentation:
(https://webcourse.cs.technion.ac.il/236370/Spring2009/ho/WCFiles/OpenMPLecture.ppt)

OpenMP

A language extension with constructs for parallel programming:

- Critical sections, atomic access, private variables, barriers

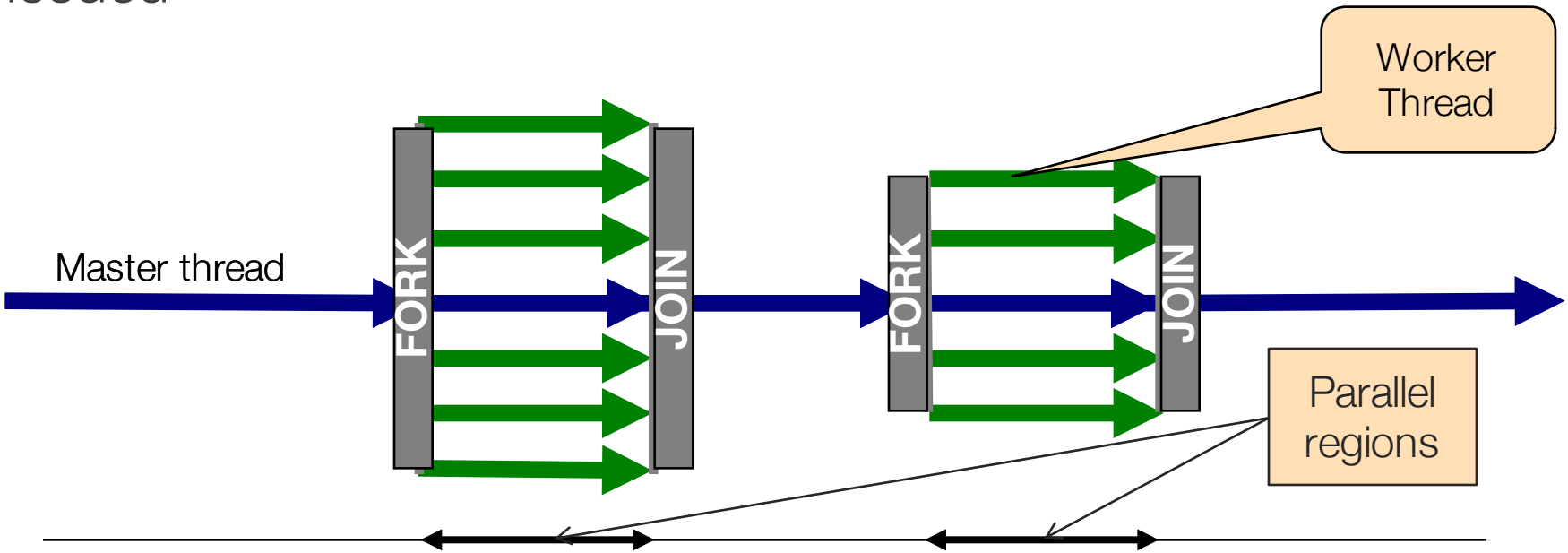
Parallelization is orthogonal to functionality

- If the compiler does not recognize OpenMP directives, the code remains functional (albeit single-threaded)

Industry standard: supported by Intel, Microsoft, IBM, HP

OpenMP execution model

Fork and Join: Master thread spawns a team of threads as needed



OpenMP memory model

Shared memory model

- Threads communicate by accessing shared variables

The sharing is defined syntactically

- Any variable that is seen by two or more threads is shared
- Any variable that is seen by one thread only is private

Race conditions possible

- Use synchronization to protect from conflicts
- Change how data is stored to minimize the synchronization

OpenMP: Work sharing example

```
answer1 = long_computation_1();
answer2 = long_computation_2();
if (answer1 != answer2) { … }
```

How to parallelize?

OpenMP: Work sharing example

```
answer1 = long_computation_1();
answer2 = long_computation_2();
if (answer1 != answer2) { … }
```

How to parallelize?

```
#pragma omp sections
{
  #pragma omp section
  answer1 = long_computation_1();
  #pragma omp section
  answer2 = long_computation_2();
}
if (answer1 != answer2) { … }
```

OpenMP: Work sharing example

Sequential code

```
for (int i=0; i<N; i++) { a[i]=b[i]+c[i]; }
```

OpenMP: Work sharing example

Sequential code

```
for (int i=0; i<N; i++) { a[i]=b[i]+c[i]; }
```

(Semi) manual parallelization

```
#pragma omp parallel
{
  int id = omp_get_thread_num();
  int nt = omp_get_num_threads();
  int i_start = id*N/nt, i_end = (id+1)*N/nt;
  for (int i=istart; i<iend; i++) { a[i]=b[i]+c[i]; }
}
```

OpenMP: Work sharing example

Sequential code

```
for (int i=0; i<N; i++) { a[i]=b[i]+c[i]; }
```

(Semi) manual parallelization

```
#pragma omp parallel
{
  int id = omp_get_thread_num();
  int nt = omp_get_num_threads();
  int i_start = id*N/nt, i_end = (id+1)*N/nt;
  for (int i=istart; i<iend; i++) { a[i]=b[i]+c[i]; }
}
```

- Launch *nt* threads
- Each thread uses *id* and *nt* variables to operate on a different segment of the arrays

OpenMP: Work sharing example

Sequential code

```
for (int i=0; i<N; i++) { a[i]=b[i]+c[i]; }
```

(Semi) manual parallelization

```
#pragma omp parallel
{
  int id = omp_get_thread_num();
  int nt = omp_get_num_
  i<
}
```

Automatic parallelizat
the for loop using
#parallel for

```
#pragma omp pa    lel
   gma omp for   chedule   tatic)
  for (int i=0; i<N; i++) { a[i]=b[i]+c[i]; }
}
```

Comparison:
var *op* last, where
op: <, >, <=, >=

Increment:
var++, var--,
var += incr, var -= incr

Initialization:
var = init

One signed variable in the loop ("i")

Challenges of #parallel for

Load balancing

- If all iterations execute at the same speed, the processors are used optimally
- If some iterations are faster, some processors may get idle, reducing the speedup
- We don't always know distribution of work, may need to re-distribute dynamically

Granularity

- Thread creation and synchronization takes time
- *Assigning work to threads on per-iteration resolution may take more time than the execution itself*
- Need to coalesce the work to coarse chunks to overcome the threading overhead

Trade-off between load balancing and granularity of parallelism

Schedule: controlling work distribution

schedule(static [, chunksize])

- Default: chunks of approximately equivalent size, one to each thread
- If more chunks than threads: assigned in round-robin to the threads
- Why might want to use chunks of different size?

schedule(dynamic [, chunksize])

- Threads receive chunk assignments dynamically
- Default chunk size = 1

schedule(guided [, chunksize])

- Start with large chunks
- Threads receive chunks dynamically. Chunk size reduces exponentially, down to chunksize

OpenMP: Data Environment

Shared Memory programming model

- Most variables (including locals) are shared by threads

```
{
   int sum = 0;
   #pragma omp parallel for
   for (int i=0; i<N; i++) sum += i;
}
```

- Global variables are shared

Some variables can be private

- Variables inside the statement block
- Variables in the called functions
- Variables can be explicitly declared as private

Overriding storage attributes

private:

- A copy of the variable is created for each thread
- There is no connection between original variable and private copies
- Can achieve same using variables inside { }

firstprivate:

- Same, but the initial value of the variable is copied from the main copy

lastprivate:

- Same, but the last value of the variable is copied to the main copy

```
int i;
#pragma omp parallel for private(i)
for (i=0; i<n; i++) { … }
```

```
int idx=1;
int x = 10;
#pragma omp parallel for \
  firsprivate(x) lastprivate(idx)
for (i=0; i<n; i++) {
   if (data[i] == x)
       idx = i;
}
```

Reduction

```
for (j=0; j<N; j++) {
  sum = sum + a[j]*b[j];
}
```

How to parallelize this code?

- sum is not private, but accessing it atomically is too expensive
- Have a private copy of sum in each thread, then add them up

Use the reduction clause

```
#pragma omp parallel for reduction(+: sum)
```

- Any associative operator could be used: +, -, ||, |, *, etc
- The private value is initialized automatically (to 0, 1, ~0 …)

#pragma omp reduction

```
float dot_prod(float* a, float* b, int N)
{
  float sum = 0.0;
#pragma omp parallel for reduction(+:sum)
  for(int i = 0; i < N; i++) {
    sum += a[i] * b[i];
  }
  return sum;
}
```

Summary

OpenMP: A framework for code parallelization

- Available for C++ and FORTRAN
- Provides control on parallelism
- Implementations from a wide selection of vendors

Relatively easy to use

- Write (and debug!) code first, parallelize later
- Parallelization can be incremental
- Parallelization can be turned off at runtime or compile time
- Code is still correct for a serial machine

Cilk vs OpenMP

Cilk:

- Simpler
- More natural for unstructured programs

OpenMP:

- Provide more control to developer (e.g., set “cunck” size)
- More natural to parallelize for() statemetns
- Wider support from more vendors; also extension for FORTRANT

“If your code looks like a sequence of parallelizable Fortran-style loops, OpenMP will likely give good speedups. If your control structures are more involved, in particular, involving nested parallelism, you may find that OpenMP isn’t quite up to the job” -- http://www.cilk.com/multicore-blog/bid/8583/Comparing-Cilk-and-OpenMP