Lecture 15-16: Parallel Programming with Cilk

Concurrent and Multicore Programming

Department of Computer Science and Engineering
Yonghong Yan
yan@oakland.edu
www.secs.oakland.edu/~yan
Topics (Part 1)

• Introduction
• Principles of parallel algorithm design (Chapter 3)
• Programming on shared memory system (Chapter 7)
  – OpenMP
  – Cilk/Cilkplus
  – PThread, mutual exclusion, locks, synchronizations
• Analysis of parallel program executions (Chapter 5)
  – Performance Metrics for Parallel Systems
    • Execution Time, Overhead, Speedup, Efficiency, Cost
  – Scalability of Parallel Systems
  – Use of performance tools
Shared Memory Systems: Multicore and Multisocket Systems

SMP - Symmetric Multiprocessor System

NUMA Architecture
Threading on Shared Memory Systems

• Employ parallelism to compute on shared data
  – boost performance on a fixed memory footprint (strong scaling)

• Useful for hiding latency
  – e.g. latency due to I/O, memory latency, communication latency

• Useful for scheduling and load balancing
  – especially for dynamic concurrency

• Relatively easy to program
  – easier than message-passing? you be the judge!
Programming Models on Shared Memory System

• Library-based models
  – All data are shared
  – Pthreads, Intel Threading Building Blocks, Java Concurrency, Boost, Microsoft .Net Task Parallel Library

• Directive-based models, e.g., OpenMP
  – shared and private data
  – pragma syntax simplifies thread creation and synchronization

• Programming languages
  – CilkPlus (Intel, GCC), and MIT Cilk
  – CUDA (NVIDIA)
  – OpenCL
At last month's meeting of the C standard committee, WG14 decided to form a study group to **produce a proposal for language extensions for C to simplify parallel programming**. This proposal is expected to **combine the best ideas from Cilk and OpenMP**, two of the most widely-used and well-established parallel language extensions for the C language family.

As the chair of this new study group, named CPLEX (C Parallel Language Extensions), I am announcing its organizational meeting:

June 17, 2013 10:00 AM PDT, 2 hours

Interested parties should join the group's mailing list, to which further information will be sent:

http://www.open-std.org/mailman/listinfo/cplex

Questions can be sent to that list, and/or to me directly.

Clark Nelson
Intel Corporation
clark.nelson@intel.com

Vice chair, PL22.16 (ANSI C++ standard committee)
Chair, SG10 (WG21 study group for C++ feature-testing)
Chair, CPLEX
(WG14 study group for C parallel language extensions)
OpenMP

- Identify static mapping and scheduling of tasks and cores
  - Before tasking
- No need to create thread manually
- Sequential code migration to parallel code by inserting directives
- Optimization for memory and synchronization are the key
  - Reduce memory contention and parallelism overhead

- Users achieve both problem decomposition into tasks and mapping tasks to hardware
  - OpenMP worksharing 1:1 mapping
Outline for Cilk/Cilkplus

- Introduction and Basic Cilk Programming
  - Cilk Work-stealing Scheduler
  - Implementation Strategies
  - Performance Analysis
  - Scheduling Performance Analysis
  - More Examples
Cilk/Cilkplus Summary

• A simpler model for writing parallel programs
  – Focusing on problem decomposition
    • What computation can be performed in parallel
  – Runtime perform the mapping

• Extends C/C++ with two main keywords → tasking
  – spawn: invoke a function (potentially) in parallel
  – sync: wait for a procedure’s spawned functions to finish

• Faithful language extension
  – if Cilk/Cilkplus keywords are elided → C/C++ program semantics

• The idea has been adopted by OpenMP with task
  – omp task
  – omp taskwait
Availability

• Cilk and Cilkplus
  – Cilk is originally developed by MIT Charles E. Leiserson
    • http://supertech.csail.mit.edu/cilk/
  – Cilkplus is commercialized now from Intel: cilk_spawn and cilk_sync
    • Added cilk_for, parallel execution of a for loop

• Availability
  – MIT Cilk
  – Intel compilers, GCC 4.9

• lennon.secs.oakland.edu
Cilk Example

Fibonacci sequence

\[
\begin{array}{cccccccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 \\
0 & +1 & +1 & +2 & +3 & +5 & +8 & +13 & 21 & 34 & 55 & 89 & 144 & 233 & 377 & 610 & 987 \\
\end{array}
\]

- Computing Fibonacci recursively

```c
int fib(int n) {
    if (n < 2) return n;
    else {
        int n1, n2;
        n1 = fib(n-1);
        n2 = fib(n-2);
        return (n1 + n2);
    }
}
```

https://en.wikipedia.org/wiki/Fibonacci_number
A Cilk program’s **serial elision** is always a legal implementation of Cilk semantics. Cilk provides *no* new data types.
Basic Cilk Keywords

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);
    }
}

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.
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);
    }
}

Example: fib(4)

The computation dag unfolds dynamically.
Mapping Tasks to Hardware

• Cilk allows the programmer to express *potential* parallelism in an application.
  – Many tasks

• The Cilk *scheduler* maps Cilk tasks onto processors dynamically at runtime
  – A thread in this context is a PE
Outline for Cilk/Cilkplus

• Introduction and Basic Cilk Programming

» Cilk Work-stealing Scheduler

• Implementation Strategies

• Performance Analysis

• Scheduling Performance Analysis

• More Examples
Scheduling Tasks in Cilk

- **Lazy parallelism**
  - Put off work for parallel execution until necessary
    - E.g. no need for parallel execution when no enough PEs

- **Work-stealing**
  - Multiple PEs share work (tasks)
    - A PE looks for work in other PEs when it becomes idle
  - Any PE can create work (tasks) via spawn

**Possible Execution:**
- thread 1 begins
- thread 2 steals from 1
- thread 3 steals from 1 etc...
Cilk’s Work-Stealing Scheduler

• Each PE maintains a **work deque** of ready tasks, and it manipulates the bottom of the deque like a stack.
  
  – Push and pop

![Diagram showing the work deque and PEs]
Cilk’s Work-Stealing Scheduler

- Each PE maintains a *work deque* of ready tasks, and it manipulates the bottom of the deque like a stack.
  - Push and pop

```
[Diagram showing the process of work deque manipulation with an example of push and pop operations.]
```
Cilk’s Work-Stealing Scheduler

- Each PE maintains a work deque of ready tasks, and it manipulates the bottom of the deque like a stack.
  - Push and pop
Cilk’s Work-Stealing Scheduler

• Each PE maintains a work deque of ready tasks, and it manipulates the bottom of the deque like a stack.
  – Push and pop

Return!
Cilk’s Work-Stealing Scheduler

- Each PE maintains a work deque of ready tasks, and it manipulates the bottom of the deque like a stack.
  - Push and pop

When a processor runs out of work, it steals a task from the top of a random victim’s deque.

Cilk’s Work-Stealing Scheduler

- Each PE maintains a work deque of ready tasks, and it manipulates the bottom of the deque like a stack.
  - Push and pop

When a processor runs out of work, it steals a task from the top of a random victim’s deque.
Cilk’s Work-Stealing Scheduler

• Each PE maintains a work deque of ready tasks, and it manipulates the bottom of the deque like a stack.
  – Push and pop

When a processor runs out of work, it steals a task from the top of a random victim’s deque.
Cilk’s Work-Stealing Scheduler

- Each PE maintains a *work deque* of ready tasks, and it manipulates the bottom of the deque like a stack.
  - Push and pop

When a processor runs out of work, it *steals* a task from the top of a *random* victim’s deque.
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);
    }
}
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);
    }
}
Workstealing State on both Program Stack and Deque

- At a fork point, add tasks to the tail of the current worker’s deque and execute the other task.
- If idle, steal work from the head of a random other worker’s deque.
- When at an incomplete join point, pop work off the tail of the worker’s own deque (reverse fork).
- If worker’s own deque is empty, either stall at join, or do a random steal.

Pablo Halpern, 2015 (CC BY 4.0)
Child-Stealing

• At a spawn, the child task in pushed onto the worker’s deque.
  – A task data structure is allocated on the heap
  – Everything needed to run the child is stored in the task data structure
  – A pointer to the task data structure is pushed onto the deque
• The worker then executes the fork continuation immediately.

• An idle worker can steal the child task.
• If the child task is not stolen, it is run by the original worker when it reaches the join point.
• *Typically*, the scheduler stalls at the join point if there are stolen children that have not completed.
Continuation Stealing

• At a spawn, the continuation in pushed onto the worker’s deque.
  – Registers are saved on the stack.
  – A pointer to the current stack frame is pushed onto the deque
• The worker then executes the child immediately, as if it were a normal call.
• An idle worker can steal the continuation task.
• Upon completing the child, if the continuation (parent) has not been stolen, the original worker continues as if returning from a normal function call.
• The join continuation is run by whichever worker completes its task last.
  – Typically, no worker stalls at the join point.
  – The worker running after the join might be different than the one entering it.
Advantages of Child stealing over continuation
Stealing

Both are types of work stealing. Continuation stealing has a number of practical advantages, however:

• Child stealing libraries can be implemented without special compiler support; continuation stealing typically requires compiler support.

• At each fork and spawn point, a continuation stealing implementation might switch to a different worker thread, confusing code that depends on thread-local storage.
Advantages of continuation stealing over Child Stealing

Conversely, continuation stealing has many theoretical advantages of continuation stealing:

• Queue size bounded by recursion depth & stack space bound to $P$ times serial stack usage vs. unbounded queue size for child stealing.

• On a single worker, continuation stealing produces identical execution to serial code; child stealing produces a scrambled execution order.

• Naturally lends itself to non-stalling join points making it closer to an ideal greedy scheduler.

• Certain features are easier to implement efficiently on top of a continuation-stealing scheduler, for example: associative reductions.
Advantages of continuation stealing over Child Stealing

Conversely continuation stealing has many theoretical advantages of child stealing:

• Queue size bounded by recursion depth & stack space bound to \( P \) times serial stack usage vs. unbounded queue size for child stealing.

• On a single worker, continuation stealing produces identical execution to serial code; child stealing produces a scrambled execution order.

• Naturally lends itself to non-stalling join points making it closer to an ideal greedy scheduler.

• Certain features are easier to implement efficiently on top of a continuation-stealing scheduler, for example: associative reductions.
Outline for Cilk/Cilkplus

• Introduction and Basic Cilk Programming
• Cilk Work-stealing Scheduler

Implementation Strategies
• Performance Analysis
• Scheduling Performance Analysis
• More Examples
Compiling \texttt{spawn} — Fast Clone

\begin{itemize}
  \item \texttt{Cilk source}
    \begin{align*}
    x &= \texttt{spawn fib(n-1)}; \\
    \text{frame->entry} &= 1; \\
    \text{frame->n} &= n; \\
    \text{push(frame)};
    \end{align*}

  \item \texttt{C post-source}
    \begin{align*}
    x &= \texttt{fib(n-1)}; \\
    \text{if (pop()==FAILURE)} \{ \\
    \quad \text{frame->x} &= x; \\
    \quad \text{frame->join}--; \\
    \quad h \text{ clean up \\
    \quad return to scheduler i}
    \}
    \end{align*}
\end{itemize}

\begin{itemize}
  \item \texttt{Cilk deque}
    \begin{itemize}
    \item suspend parent
    \item run child
    \item resume parent remotely
    \end{itemize}
\end{itemize}
Compiling **sync** — Fast Clone

Cilk source

```
 sync;
cilk2c
```

C post-source

No synchronization overhead in the fast clone!
void fib_slow(fib_frame *frame) {
    int n, x, y;
    switch (frame->entry) {
        case 1: goto L1;
        case 2: goto L2;
        case 3: goto L3;
    }
    frame->entry = 1;
    frame->n = n;
    push(frame);
    x = fib(n-1);
    if (pop()==FAILURE) {
        frame->x = x;
        frame->join--;
        h clean up &
        return to scheduler i
    }
    if (0) {
        L1:;
        n = frame->n;
    }
}

restore
program
counter

same
as fast
clone

restore local
variables
if resuming

continue

Cilk
deque
Project Accounts

• On orion.ec.oakland.edu
  – Need VPN to access from home

• Account is the same as your netid
  – Password: <first four letters of your netid>1234
  – Change it the first time you login

• Follow development setup steps to clone the OpenMP runtime repo and examples repo
Outline for Cilk/Cilkplus

• Introduction and Basic Cilk Programming
• Cilk Work-stealing Scheduler
• Implementation Strategies

† Performance Analysis

• Scheduling Performance Analysis
• More Examples
The dag $G = (V, E)$ represents a parallel instruction stream.

Each vertex $v$ of $V$ represents a (Cilk) task: a maximal sequence of instructions not containing parallel control (spawn, sync, return).

Every edge $e$ of $E$ is either a spawn edge, a return edge, or a continue edge.
Algorithmic Complexity Analysis

\[ T_P = \text{execution time on } P \text{ processors} \]

- **Computation graph abstraction:**
  - node = arbitrary sequential computation
  - edge = dependence (successor node can only execute after predecessor node has completed)
  - Directed Acyclic Graph (DAG)

- **Processor abstraction:**
  - \( P \) identical processors
  - each processor executes one node at a time
Algorithmic Complexity Analysis

\[ T_P = \text{execution time on } P \text{ processors} \]
Algorithmic Complexity Analysis

\[ T_p = \text{execution time on } P \text{ processors} \]

\[ T_1 = \text{work} \]
Algorithmic Complexity Analysis

\[ T_P = \text{execution time on } P \text{ processors} \]

\[ T_1 = \text{work} \]

\[ T_\infty = \text{span*} \]

* Also called critical-path length or computational depth.
Algorithmic Complexity Analysis

\[ T_P = \text{execution time on } P \text{ processors} \]

\[ T_1 = \text{work} \]

\[ T_\infty = \text{span}^* \]

\[ \text{LOWER BOUNDS} \]

- \[ T_P \geq T_1 / P \]
- \[ T_P \geq T_\infty \]

*Also called critical-path length or computational depth.*
**Speedup**

**Definition:** \( T_1/T_P = \text{speedup} \) on \( P \) processors.

If \( T_1/T_P = \Theta(P) \), we have *linear speedup*; 
\( = P \), we have *perfect linear speedup*; 
\( > P \), we have *superlinear speedup*, which is not possible in our model, because of the lower bound \( T_P \geq T_1/P \).
Parallelism and Parallel Slackness

• We have the lower bound $T_P \geq T_\infty$ and $T_P \geq T_1/P$

• The maximum possible speedup given $T_\infty$ and $T_1$, i.e. the parallelism
  – Independent of $P$, only depend on the graph
    \[ P = \frac{T_1}{T_\infty} \]

• Parallel slackness (Efficiency) as the ratio
  \[ \frac{(T_1/T_\infty)}{P} \]
  – The larger the efficiency, the less the impact of $T_\infty$ on performance
Example: $\text{fib}(4)$

Assume for simplicity that each Cilk task in $\text{fib}(\cdot)$ takes unit time to execute.

**Work:** $T_1 = 17$

**Span:** $T_\infty = 8$

**Parallelism:** $\frac{T_0}{T_\infty} = 2.125$

*Using many more than 2 processors makes little sense.*
Parallelizing Vector Addition

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

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);
    }
}

Parallelization strategy:
1. Convert loops to recursion.
Parallelizing Vector Addition

### C

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

### Cilk

```c
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);
    }
sync;
}
```

**Parallelization strategy:**

1. Convert loops to recursion.
2. Insert Cilk keywords.

**Side benefit:**

divide and conquer is generally good for caches!
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;
    }
}
Vector Addition Analysis

To add two vectors of length $n$, where BASE = $\Theta(1)$:

**Work:** $T_1 = \, ?$  \hspace{1cm}  $\Theta(n)$

**Span:** $T_\infty = \, ?$  \hspace{1cm}  $\Theta(\log n)$

**Parallelism:** $T_1/T_\infty = \, ?$  \hspace{1cm}  $\Theta(n/\log n)$

Vector Addition Analysis

To add two vectors of length $n$, where BASE = $\Theta(1)$:

**Work:** $T_1 = \, ?$  \hspace{1cm}  $\Theta(n)$

**Span:** $T_\infty = \, ?$  \hspace{1cm}  $\Theta(\log n)$

**Parallelism:** $T_1/T_\infty = \, ?$  \hspace{1cm}  $\Theta(n/\log n)$
Outline for Cilk/Cilkplus

• Introduction and Basic Cilk Programming
• Cilk Work-stealing Scheduler
• Implementation Strategies
• Performance Analysis
  ➡️ Scheduling Performance Analysis
• More Examples
Analysis: Greedy Scheduling

**IDEA:** Do as much as possible on every step.

**Definition:** A task is *ready* if all its predecessors have *executed.*
Greedy Scheduling

**IDEA:** Do as much as possible on every step.

**Definition:** A task is *ready* if all its predecessors have *executed*.

**Complete step**
- $\geq P$ tasks ready.
- Run any $P$. 

\[ P = 3 \]
**Greedy Scheduling**

**IDEA:** Do as much as possible on every step.

**Definition:** A task is *ready* if all its predecessors have *executed*.

**Complete step**
- $\geq P$ tasks ready.
- Run any on $P$.

**Incomplete step**
- $< P$ tasks ready.
- Run all of them.
Theorem [Graham ’68 & Brent ’75]. Any greedy scheduler achieves

\[ T_P \leq T_1/P + T_\infty \]

**Proof.**
- # complete steps \( \leq T_1/P \), since each complete step performs \( P \) work.
- # incomplete steps \( \leq T_\infty \), since each incomplete step reduces the span of the unexecuted dag by 1.
**Performance of Work-Stealing**

**Theorem**: On $P$ processors, Cilk’s work-stealing scheduler achieves an expected running time of

$$T_p = \frac{T_1}{P} + O(T_\infty)$$

- **work term**
- **Critical path term**
Critical Path Overhead

- Critical path overhead = smallest constant $C_\infty$ such that

\[
T_p \leq \frac{T_1}{P} + c_\infty T_\infty
\]

Let $\overline{P} = \frac{T_1}{T_\infty} = \frac{P}{\infty}$ = parallelism = max speedup on $\infty$ processors

\[
T_P \leq \left( \frac{T_1}{T_\infty P} + c_\infty \right) T_\infty = \left( \frac{\overline{P}}{P} + c_\infty \right) T_\infty
\]

Parallel slackness assumption

\[
\frac{\overline{P}}{P} >> c_\infty
\]

thus

\[
\frac{T_1}{P} >> c_\infty T_\infty
\]

$T_p \approx \frac{T_1}{P}$ linear speedup

“critical path overhead has little effect on performance when sufficient parallel slackness exists”
Work Overhead

\[ c_1 = \frac{T_1}{T_s} \quad \text{work overhead} \]

\[ T_p \leq c_1 \frac{T_s}{P} + c_\infty T_\infty \]

“Minimize work overhead \((c_1)\) at the expense of a larger critical path overhead \((c_\infty)\), because work overhead has a more direct impact on performance”

\[ T_p \approx c_1 \frac{T_s}{P} \quad \text{assuming parallel slackness} \]

You can reduce \(C_1\) by increasing the granularity of parallel work
**Breakdown of Work Overhead**

- **MIPS R10000**
  - 115ns
  - 27ns
- **UltraSPARC I**
  - 113ns
  - 115ns
- **Pentium Pro**
  - 78ns
- **Alpha 21164**
  - 27ns

**Benchmark:** `fib` on one processor.

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.
Outline for Cilk/Cilkplus

• Introduction and Basic Cilk Programming
• Cilk Work-stealing Scheduler
• Implementation Strategies
• Performance Analysis
• Scheduling Performance Analysis

More Examples
**Square-Matrix Multiplication**

\[
\begin{pmatrix}
c_{11} & c_{12} & \cdots & c_{1n} \\
c_{21} & c_{22} & \cdots & c_{2n} \\
\vdots & \vdots & \ddots & \vdots \\
c_{n1} & c_{n2} & \cdots & c_{nn}
\end{pmatrix}
= \begin{pmatrix}
a_{11} & a_{12} & \cdots & a_{1n} \\
a_{21} & a_{22} & \cdots & a_{2n} \\
\vdots & \vdots & \ddots & \vdots \\
a_{n1} & a_{n2} & \cdots & a_{nn}
\end{pmatrix}
\times
\begin{pmatrix}
b_{11} & b_{12} & \cdots & b_{1n} \\
b_{21} & b_{22} & \cdots & b_{2n} \\
\vdots & \vdots & \ddots & \vdots \\
b_{n1} & b_{n2} & \cdots & b_{nn}
\end{pmatrix}
\]

\[c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}\]

Assume for simplicity that \(n = 2^k\).
Recursive Matrix Multiplication

Divide and conquer —

\[
\begin{pmatrix}
C_{11} & C_{12} \\
C_{21} & C_{22}
\end{pmatrix} =
\begin{pmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{pmatrix}
\times
\begin{pmatrix}
B_{11} & B_{12} \\
B_{21} & B_{22}
\end{pmatrix}
\]

\[
= \begin{pmatrix}
A_{11}B_{11} & A_{11}B_{12} \\
A_{21}B_{11} & A_{21}B_{12}
\end{pmatrix}
+ \begin{pmatrix}
A_{12}B_{21} & A_{12}B_{22} \\
A_{22}B_{21} & A_{22}B_{22}
\end{pmatrix}
\]

8 multiplications of \((n/2) \times (n/2)\) matrices.
1 addition of \(n \times n\) matrices.
Matrix Multiplication

    // C = C + A * B
    h base case & partition matrices i
    spawn MultA(C11,A11,B11,n/2);
    spawn MultA(C12,A11,B12,n/2);
    spawn MultA(C22,A21,B12,n/2);
    spawn MultA(C21,A21,B11,n/2);
    sync;
    spawn MultA(C21,A22,B21,n/2);
    spawn MultA(C22,A22,B22,n/2);
    spawn MultA(C12,A12,B22,n/2);
    spawn MultA(C11,A12,B21,n/2);
    sync;
    return;
}
Work of Multiply

    // C = C + A * B
    // base case & partition matrices
    spawn MultA(C11, A11, B11, n/2);
    spawn MultA(C12, A11, B12, n/2);
    spawn MultA(C22, A21, B12, n/2);
    spawn MultA(C21, A21, B11, n/2);
    sync;
    spawn MultA(C21, A22, B21, n/2);
    spawn MultA(C22, A22, B22, n/2);
    spawn MultA(C12, A12, B22, n/2);
    spawn MultA(C11, A12, B21, n/2);
    sync;
    return;
}

Work: \[ T_1(n) = \Theta(n^3) \]
Span of Multiply

\[ T_{\infty}(n) = 2 \cdot T_{\infty}(n/2) + \Theta(1) \]
\[ = \Theta(n) \]

**Parallelism:** \[ T_1/T_{\infty} = \Theta(n^3) / \Theta(n) = \Theta(n^2) \]
void Merge(int *C, int *A, int *B, int na, int nb) {
    while (na>0 && nb>0) {
        if (*A <= *B) {
            *C++ = *A++; na--;
        } else {
            *C++ = *B++; nb--;
        }
    }
    while (na>0) {
        *C++ = *A++; na--;
    }
    while (nb>0) {
        *C++ = *B++; nb--;
    }
}

Time to merge $n$ elements = $\Theta(n)$. 
cilk void MergeSort(int *B, int *A, int n) {
    if (n==1) {
        B[0] = A[0];
    } else {
        int *C;
        C = (int*) Cilkalloca(n*sizeof(int));
        spawn MergeSort(C, A, n/2);
        spawn MergeSort(C+n/2, A+n/2, n-n/2);
        sync;
        Merge(B, C, C+n/2, n/2, n-n/2);
    }
}
Work of Merge Sort

cilk void MergeSort(int *B, int *A, int n) {
    if (n==1) {
        B[0] = A[0];
    }
    else {
        int *C;
        C = (int*) Cilk_alloca(n*sizeof(int));
        spawn MergeSort(C, A, n/2);
        spawn MergeSort(C+n/2, A+n/2, n-n/2);
        sync;
        Merge(B, C, C+n/2, n/2, n-n/2);
    }
}

Work: \[ T_1(n) = 2 \cdot T_1(n/2) + \Theta(n) \]

\[ = \Theta(n \log n) \]
Span of Merge Sort

cilk void MergeSort(int *B, int *A, int n) {
  if (n==1) {
    B[0] = A[0];
  } else {
    int *C;
    C = (int*) Cilk_alloca(n*sizeof(int));
    spawn MergeSort(C, A, n/2);
    spawn MergeSort(C+n/2, A+n/2, n-n/2);
    sync;
    Merge(B, C, C+n/2, n/2, n-n/2);
  }
}

Span: \[ T_\infty(n) = T_\infty(n/2) + \Theta(n) = \Theta(n) \]

Parallelism: \[ \frac{T_1(n)}{T_\infty(n)} = \Theta(\lg n) \]
Tableau Construction

**Problem:** Fill in an $n \times n$ tableau $A$, where


<table>
<thead>
<tr>
<th></th>
<th>00</th>
<th>01</th>
<th>02</th>
<th>03</th>
<th>04</th>
<th>05</th>
<th>06</th>
<th>07</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
<td>37</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>41</td>
<td>42</td>
<td>43</td>
<td>44</td>
<td>45</td>
<td>46</td>
<td>47</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>51</td>
<td>52</td>
<td>53</td>
<td>54</td>
<td>55</td>
<td>56</td>
<td>57</td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>61</td>
<td>62</td>
<td>63</td>
<td>64</td>
<td>65</td>
<td>66</td>
<td>67</td>
<td></td>
</tr>
<tr>
<td>70</td>
<td>71</td>
<td>72</td>
<td>73</td>
<td>74</td>
<td>75</td>
<td>76</td>
<td>77</td>
<td></td>
</tr>
</tbody>
</table>

**Dynamic programming**

- Longest common subsequence
- Edit distance
- Time warping

**Work:** $\Theta(n^2)$. 
Recursive Construction

Cilk code

- spawn I;
- sync;
- spawn II;
- sync;
- spawn III;
- sync;
- spawn IV;
- sync;
Recursive Construction

Work: $T_1(n) = 4T_1(n/2) + \Theta(1) = \Theta(n^2)$
Recursive Construction

Span: \( T_\infty(n) = 3T_\infty(n/2) + \Theta(1) = \Theta(n^{\lg 3}) \)

Parallelism: \( \frac{T_1(n)}{T_\infty(n)} \approx \Theta(n^{0.42}) \)

Cilk code

\begin{align*}
\text{spawn I; sync;} \\
\text{spawn II; sync;} \\
\text{spawn III; sync;} \\
\text{spawn IV; sync;} \\
\end{align*}
A More-Parallel Construction

<table>
<thead>
<tr>
<th></th>
<th>I</th>
<th>II</th>
<th>IV</th>
</tr>
</thead>
<tbody>
<tr>
<td>III</td>
<td>V</td>
<td>VII</td>
<td></td>
</tr>
<tr>
<td>VI</td>
<td>VIII</td>
<td>IX</td>
<td></td>
</tr>
</tbody>
</table>

```
spawn I;
spawn sync;
spawn II;
spawn sync;
spawn III;
spawn sync;
spawn IV;
spawn V;
spawn VI;
spawn VII;
spawn VIII;
spawn IX;
spawn sync;
```
A More-Parallel Construction

Work: $T_1(n) = 9T_1(n/3) + \Theta(1)$

$= \Theta(n^2)$
$\textit{Span: } T_\infty(n) = 5T_\infty(n/3) + \Theta(1) = \Theta(n \log_3 5)$
Analysis of Revised Construction

**Work:** $T_1(n) = \Theta(n^2)$

**Span:** $T_\infty(n) = \Theta(n^{\log_35})$

\[ \approx \Theta(n^{1.46}) \]

**Parallelism:**

\[
\frac{T_1(n)}{T_\infty(n)} \approx \Theta(n^{0.54})
\]

More parallel by a factor of

\[
\frac{\Theta(n^{0.54})}{\Theta(n^{0.42})} = \Theta(n^{0.12})
\]
References

- Intel Cilk++ Programmer’s Guide. Document # 322581-001US.