Cilk

A Forerunning Language for Parallel Programming

Olivier Aumage – STORM Team
Inria – LaBRI

Technological Development’s Tuesdays
Introduction

The two “all-time” goals in parallel programming

- Programming parallel applications
  - Easily

- Running parallel applications
  - Efficiently

The Cilk language and framework played an anticipative role in reaching these goals for some classes of applications
Cilk

Cilk in a Few Words

- A programming environment
  - A language and compiler: keyword-based extension of C
  - An execution model and a run-time system

- Developed at the MIT
  - Supertech Research Group
  - Charles E. Leiserson’s team
  - Mid-90’s
History

- **Academic Era — Cilk**
  - 1994: Cilk 1

- **Start-up Era — Cilk++**
  - 2006: Launch of “Cilk Arts” company
  - 2008: Cilk++ version 1.0

- **Intel Era — Cilk Plus**
  - 2009: Intel acquires Cilk Arts
  - 2010: Intel Cilk Plus released as part of the Intel C++ Compiler
  - 2012: Release of the Cilk Plus support for the GNU GCC Compiler, implemented by Intel
Context

Middle of nineties

Hardware

- SMP: Symmetric Multi-Processors
- Need for parallel programming models
Context

Middle of nineties

Hardware
- SMP: Symmetric Multi-Processors
- Need for parallel programming models

Software
- Notion of threads: concurrent processing contexts within single process
- How to efficiently/easily express application parallelism using threads?
Context

Middle of nineties

Hardware
- SMP: Symmetric Multi-Processors
- Need for parallel programming models

Software
- Notion of threads: concurrent processing contexts within single process
- How to efficiently/easily express application parallelism using threads?

Program Easily?
- Parallel program quickly derived from sequential program
- Concurrency expressed safely (correctness, consistency)
Context

Middle of nineties

Hardware
- SMP: Symmetric Multi-Processors
- Need for parallel programming models

Software
- Notion of threads: concurrent processing contexts within single process
- How to efficiently/easily express application parallelism using threads?

Program **Easily**?
- Parallel program quickly derived from sequential program
- Concurrency expressed safely (correctness, consistency)

Run **Efficiently**?
- No over/under-subscription
- Load-balancing
- Low overhead
Context

Nowadays

Hardware

- SMP: Symmetric Multi-Processors
- *Multi-core processors*
- Hardware multi-threading (SMT, Hyperthreading, etc.)
- Even greater need for parallel programming models!
Context

Nowadays

Hardware
- SMP: Symmetric Multi-Processors
- *Multi-core processors*
- Hardware multi-threading (SMT, Hyperthreading, etc.)
- Even greater need for parallel programming models!

Revived interest for Cilk-like approaches
2

Concept and Ideas
Concept and Ideas

- C extension
- Task Parallelism paradigm
- Work Stealing paradigm
- THE Algorithm
C Extension

Cilk is a faithful extension of C

- Keyword based language
  - Per opposition to pragma based languages
C Extension

Cilk is a faithful extension of C

- Keyword based language
  - Per opposition to pragma based languages

- Main keywords (original Cilk):
  - cilk: declaration of a potentially parallel routine
  - spawn: launch of a potentially parallel routine
  - sync: wait for completion of launched routines
  - inlet: special function to aggregate results (reduction)

```c
#include <cilk.h>

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;
    }
}
```
C Extension

Cilk is a faithful extension of C

- **Keyword based language**
  - Per opposition to pragma based languages
- **Main keywords (original Cilk):**
  - `cilk`: declaration of a potentially parallel routine
  - `spawn`: launch of a potentially parallel routine
  - `sync`: wait for completion of launched routines
  - `inlet`: special function to aggregate results (reduction)

- **A faithful extension of C**
  - The C *elision* of a Cilk program is a valid C program

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

Cilk is a faithful extension of C

- Keyword based language
  - Per opposition to pragma based languages
- Main keywords (original Cilk):
  - `cilk`: declaration of a potentially parallel routine
  - `spawn`: launch of a potentially parallel routine
  - `sync`: wait for completion of launched routines
  - `inlet`: special function to aggregate results (reduction)
- A faithful extension of C
  - The C *elision* of a Cilk program is a valid C program

```c
#include <cilk.h>
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;
  }
}
```
Task Parallelism Paradigm

Potential Parallelism

- The programmer declares **what can be run** in parallel
- Cilk’s runtime decides **what to run** in parallel
Task Parallelism Paradigm

Potential Parallelism

- The programmer declares **what can be run** in parallel
- Cilk’s runtime decides **what to run** in parallel

Work / Worker Decoupling

- Workers: **threads** launched by Cilk
- Work: **tasks** expressed by the Application
Task Parallelism Paradigm

Potential Parallelism

- The programmer declares **what can be run** in parallel
- Cilk’s runtime decides **what to run** in parallel

Work / Worker Decoupling

- Workers: **threads** launched by Cilk
- Work: **tasks** expressed by the Application

Rationalization

- Generic worker threads instead of special purpose threads
- Externalized common parallelism management
- Externalized resource allocation policy
  - Potential parallelism vs number of concurrent contexts
Tasks

Notion of frame

- State of the current cilk function being executed
- Live local variables, function arguments
- “Program Counter” (PC)
Tasks

Notion of frame

- State of the current cilk function being executed
- Live local variables, function arguments
- “Program Counter” (PC)

A Frame is a Task
Task Lists

Notion of frame deque

- One task list per worker
- Implemented as a "deque" (doubly-ended queue of frames)
  - Head H
  - Tail T
Task Lists

Notion of frame deque

- One task list per worker
- Implemented as a “deque” (doubly-ended queue of frames)
  - Head H
  - Tail T
Task Lists

Notion of frame deque

- One task list per worker
- Implemented as a “deque” (doubly-ended queue of frames)
  - Head H
  - Tail T

A worker thread pushes/pops frames at the Tail side of its deque
Task Lists

Notion of frame deque

- One task list per worker
- Implemented as a “deque” (doubly-ended queue of frames)
  - Head H
  - Tail T

A worker thread pushes/pops frames at the Tail side of its deque
Task Lists

Notion of frame deque

- One task list per worker
- Implemented as a “deque” (doubly-ended queue of frames)
  - Head H
  - Tail T

A worker thread pushes/pops frames at the Tail side of its deque
Task Lists

Notion of frame deque

- One task list per worker
- Implemented as a “deque” (doubly-ended queue of frames)
  - Head H
  - Tail T

A worker thread pushes/pops frames at the Tail side of its deque
Task Spawn

Upon a spawn, the worker...

- ... suspends the parent function
- ... saves the state of the parent function in its frame
- ... pushes the parent frame on its task list

...before calling the spawned child function
Task Spawn

Upon a spawn, the worker...

- ... suspends the parent function
- ... saves the state of the parent function in its frame
- ... pushes the parent frame on its task list

...before calling the spawned child function

When the child function completes and returns, the worker...

- ... pops the parent frame
- ... restores the state of the parent function from its frame
- ... resumes the parent function
Task Spawn

Upon a spawn, the worker...
- ... suspends the parent function
- ... saves the state of the parent function in its frame
- ... pushes the parent frame on its task list
... before calling the spawned child function

When the child function completes and returns, the worker...
- ... pops the parent frame
- ... restores the state of the parent function from its frame
- ... resumes the parent function

This is more or less what regular functions do...
Task Spawn

Upon a spawn, the worker...

- ... suspends the parent function
- ... saves the state of the parent function in its frame
- ... pushes the parent frame on its task list

... before calling the spawned child function

When the child function completes and returns, the worker...

- ... pops the parent frame
- ... restores the state of the parent function from its frame
- ... resumes the parent function

This is more or less what regular functions do...

Where is the parallelism?
Example: Deque Management on Spawn

```c
#include <cilk.h>

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: Deque Management on Spawn

```cilk
#include <cilk.h>

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;
    }
}
```

Worker

H = T

1
2
3
4
5
6
7
8
9
10
Example: Deque Management on Spawn

```cilk
#include <cilk.h>

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: Deque Management on Spawn

1. #include <cilk.h>

2. cilk int fib (int n) {
3.     if (n < 2) {
4.         return n;
5.     } else {
6.         int x, y;
7.         x = spawn fib (n-1);
8.         y = spawn fib (n-2);
9.         sync;
10.        return x+y;
11.     }
12. }

1. Fib n, step 2
2. Fib n-1, step 2

Worker
#include `<cilk.h>`

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: Deque Management on Spawn

```c
#include <cilk.h>

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;
    }
}
```

Worker

<table>
<thead>
<tr>
<th></th>
<th>Fib n−2, step 2</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Fib n, step 2</td>
<td>H</td>
</tr>
<tr>
<td>2</td>
<td>Fib n−1, step 2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Fib n−2, step 2</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>T</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example: Deque Management on Spawn

```c
#include <cilk.h>

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;
    }
}
```

```
1 | Fib n, step 2 | H
2 | Fib n-1, step 2 |
3 | Fib n-2, step 2 |
4 | ... |
5 | Fib 2, step 2 |
6 |              |
7 |              |
8 |              |
9 |              |
10 |              |
```

Worker
Example: Deque Management on Spawn

```cilk
#include <cilk.h>

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;
    }
}
```

```
1         2         3         4         5         6         7         8         9  10
Fib n, step 2         H         Fib n-1, step 2
Fib n-2, step 2         ...         T
Worker
```
Example: Deque Management on Spawn

```c
#include <cilk.h>

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;
    }
}
```

<table>
<thead>
<tr>
<th></th>
<th>Fib n, step 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Fib n-1, step 2</td>
</tr>
<tr>
<td>3</td>
<td>Fib n-2, step 2</td>
</tr>
<tr>
<td>4</td>
<td>...</td>
</tr>
<tr>
<td>5</td>
<td>Fib 2, step 3</td>
</tr>
<tr>
<td>6</td>
<td>H</td>
</tr>
<tr>
<td>7</td>
<td>T</td>
</tr>
<tr>
<td>8</td>
<td>Worker</td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>
Parallelism

*Work Stealing* paradigm

- **Idle** workers steal work...
- ... from other worker’s queues
Parallelism

*Work Stealing* paradigm

- **Idle** workers steal work...
- ... from other worker’s queues

Work stolen as frame/task

- A thief resumes a suspended parent task...
- ... **while** a victim runs its child
Parallelism

*Work Stealing* paradigm

- **Idle** workers steal work...
- ... from other worker’s queues

Work stolen as frame/task

- A thief resumes a suspended parent task...
- ... *while* a victim runs its child

Load balancing: Idle workers steal from busy workers
Task Spawn [UPDATED]

Upon a spawn, the worker...

- ... suspends the parent function
- ... saves the state of the parent function in its frame
- ... pushes the parent frame on its task list

...before calling the spawned child function
Task Spawn [UPDATED]

Upon a spawn, the worker...

- ... suspends the parent function
- ... saves the state of the parent function in its frame
- ... pushes the parent frame on its task list
...before calling the spawned child function

When the child function completes and returns, the worker...

- ... attempts to pop the parent frame
- if it succeeds, it...
  - ... restores the state of the parent function from its frame
  - ... resumes the parent function
Work Stealing Implementation

Task lists implementation...

- Doubly-ended queue
- Head H
- Tail T

Workers push/pop work at the Tail side of their own deque.

An idle worker (thief) steals work at the Head side of another worker (victim) deque.

T ≥ H under normal conditions.
Work Stealing Implementation

Task lists implementation...

- Doubly-ended queue
- Head H
- Tail T

... with the following rules

- Workers push/pop work at the Tail side T of their own deque
- An idle worker (thief) steals work at the Head side H of another worker (victim) deque
- T ≥ H under normal conditions
Example: Work Stealing

```c
#include <cilk.h>
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;
    }
}
```

```
Fib n, step 2
Fib n-1, step 2
Fib n-2, step 2
Fib 2, step 3
```

Worker
Example: Work Stealing

```c
#include <cilk.h>

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: Work Stealing

```c
#include "cilk.h"

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;
    }
}
```

Worker

1. Fib 2, step 3
2. . .
3. Fib n−2, step 2
4. . .
5. Fib 2, step 3
6. . .
7. . .
8. . .
9. . .
10. . .

Worker
Work Stealing Implementation Discussion

Doubly-ended queue benefits

- Tail-side worker push/pop
  - Depth-first
  - Locality

- Head side steal
  - Breadth-first
  - Potentially favor big steals
  - Potentially steal less often

- Potentially rare deque interferences between worker and thieves

Low overhead, high efficiency for Divide and Conquer algorithms
Example: Conflict

```c
#include <cilk.h>

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;
  }
}
```

Diagram:
- **Worker**
  - Fib n, step 2
  - Fib n−1, step 2
  - Fib n−2, step 2
  - Fib 2, step 3
  - . . .
Example: Conflict

```c
#include <cilk.h>

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: Conflict

```c
#include <cilk.h>

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;
    }
}
```

Fib 2, step 3
. . .
Fib n−2, step 2
T
H
Worker

Example: Conflict

```c
#include <cilk.h>

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;  
    }  
}
```

Worker

Fib 2, step 3

H

T
TH(E) Algorithm

Generalities

- T.H.(E.): Tail, Head, (Exception)
- Optimization of the deque synchronization mechanism
  - Favor the potentially frequent worker pushes/pops...
  - ... over the potentially infrequent thief steals
TH(E) Algorithm

Generalities

- T.H.(E.): Tail, Head, (Exception)
- Optimization of the deque synchronization mechanism
  - Favor the potentially frequent worker pushes/pops...
  - ... over the potentially infrequent thief steals

Principles

- The deque is protected by a expensive lock L
- The thieves only modify H
  - Thieves always take the lock L before changing H
- The worker only modifies T
- Conflicts between a worker and a thief arise when...
  - ... the invariant T >= H is broken
TH(E) Algorithm

Generalities
- T.H.(E.): Tail, Head, (Exception)
- Optimization of the deque synchronization mechanism
  - Favor the potentially frequent worker pushes/pops...
  - ... over the potentially infrequent thief steals

Principles
- The deque is protected by a expensive lock L
- The thieves only modify H
  - Thieves always take the lock L before changing H
- The worker only modifies T
- Conflicts between a worker and a thief arise when...
  - ... the invariant T >= H is broken

The worker speculatively modifies T without taking the lock L!
TH(E) Algorithm

Generalities

- T.H.(E.): Tail, Head, (Exception)
- Optimization of the deque synchronization mechanism
  - Favor the potentially frequent worker pushes/pops...
  - ... over the potentially infrequent thief steals

Principles

- The deque is protected by a expensive lock L
- The thieves only modify H
  - Thieves always take the lock L before changing H
- The worker only modifies T
- Conflicts between a worker and a thief arise when...
  - ... the invariant T >= H is broken

The worker speculatively modifies T without taking the lock L!

If the speculation fails, the worker retries with the L lock taken
TH(E) Algorithm Pseudocode

```
1 /* Worker */
2 push () {
3   T++;
4 }
5
6 pop () {
7   T--;__mem_fence__();
8  if (H > T) {
9     T++;
10    lock (L);
11    T--;__mem_fence__();
12   }
13 }
14
15 /* Thief */
16 steal () {
17    lock (L);
18    H++;
19    __mem_fence__();
20    if (H > T) {
21      H--;
22      unlock (L);
23      return FAILURE;
24    }
25    unlock (L);
26    return SUCCESS;
27 }
```
Cilk’s Keywords Summary

Rehearsal before looking at some examples

- **cilk**: declaration of a potentially parallel routine
- **spawn**: launch of a potentially parallel routine
- **sync**: wait for completion of launched routines

```c
#include <cilk.h>
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;
    }
}
```
Cilk’s Keywords Summary

Rehearsal before looking at some examples

- **cilk**: declaration of a potentially parallel routine
- **spawn**: launch of a potentially parallel routine
- **sync**: wait for completion of launched routines
- **inlet**: special function to aggregate results (reduction)

```c
#include <cilk.h>

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;
    }
}
```
Cilk’s Keywords Summary

Rehearsal before looking at some examples

- **cilk**: declaration of a potentially parallel routine
- **spawn**: launch of a potentially parallel routine
- **sync**: wait for completion of launched routines
- **inlet**: special function to aggregate results (reduction)

```c
#include <cilk.h>

cilk int fib (int n) {
    int x = 0;

    inlet void acc (int res) {
        x += res;
    }

    if (n < 2) {
        return n;
    } else {
        acc (spawn fib (n-1));
        acc (spawn fib (n-2));
        sync;
        return x;
    }
}
```
Examples

- Fib
- Knapsack
- Heat
- LU
Code Generation Example

- Fib

Two-clone Strategy
- Fast clone executed by worker
  - “Streamlined” C function
- Slow clone executed by a thief after a steal
  - Post-steal logic to restore saved state
3

Cilk Plus
Intel Cilk Plus

**URL:** http://www.cilkplus.org/

**Changes**
- Supports C and C++
- No need to declare Cilk functions
Intel Cilk Plus

**URL**: http://www.cilkplus.org/

**Changes**
- Supports C and C++
- No need to declare Cilk functions

**Main keywords**
- `cilk_spawn`: similar to original Cilk’s `spawn`
- `cilk_sync`: similar to original Cilk’s `sync`
- `cilk ::reducer <...>`
  - Template parameterized with a reduction op
  - Replacement for inlets
- `cilk_for`: parallel loop
- Fortran inspired *Array Notation*
Cilk Plus Parallel Loop

Work-Stealing Loop

- cilk_for keyword
- Potentially parallel loop
- Recursively divided range
- Work-stealing load balancing

```c
int i;

for (i=0; i<N; i++) {
    f(i);
}

/* -------------- */

for (i=0; i<N; i++) {
    cilk_spawn f(i);
}

cilk_sync;

/* -------------- */

cilk_for (i=0; i<N; i++) {
    f(i);
}
```
Intel Cilk Plus Arrays

Array notation

- Inspired by Fortran
- Helps compiler auto-vectorization
  - May use SIMD instruction sets (SSE, AVX)
Intel Cilk Plus Arrays

Array notation

- Inspired by Fortran
- Helps compiler **auto-vectorization**
  - May use SIMD instruction sets (SSE, AVX)

- Syntax:

  ```
  double a[lower_bound:length:stride];
  ```
Intel Cilk Plus Arrays

Array notation

- Inspired by Fortran
- Helps compiler **auto-vectorization**
  - May use SIMD instruction sets (SSE, AVX)

Syntax:

```c
double a[lower_bound:length:stride];
```

Implicit vector loop:

```c
for (i = lower; i < (len*stride)+lower; i += stride)
   ... a[i] ...
```
Intel Cilk Plus Arrays

Array notation

- Inspired by Fortran
- Helps compiler **auto-vectorization**
  - **May** use SIMD instruction sets (SSE, AVX)
- Syntax:
  ```c
  double a[lower_bound:length:stride];
  ```
- Implicit vector loop:
  ```c
  for (i = lower; i < (len*stride)+lower; i += stride)
      ... a[i] ...
  ```
- Shortcut syntax to access all elements (static arrays only):
  ```c
  a[:]
  ```
Intel Cilk Plus Arrays

Example expressions

- 1D arrays:

1. `a[:] = 1;`
2. `a[2:4] = 3;`
3. `a[1:10:2] = 7; // set odd indices`

- Multi-dimensional arrays:

1. `c[::] = 17;`
2. `c[3][:] = a[:]; // affect an array to some row`
3. `c[:][4] = a[:]; // affect an array to some column`
4. `c[0:10:2][:] = 1; // set even rows`
Intel Cilk Plus Arrays

Example expressions

- 1D arrays:
  
  ```
  a[::] = 1;
  a[2:4] = 3;
  a[1:10:2] = 7;  // set odd indices
  ```

- Multi-dimensional arrays:
  
  ```
  c[::][::] = 17;
  c[3][::] = a[::];  // affect an array to some row
  c[::][4] = a[::];  // affect an array to some column
  c[0:10:2][::] = 1;  // set even rows
  ```
Intel Cilk Plus Arrays

Advanced example expressions

- Conditions:

```c
if (a[::] == 1) {
    b[::] = 42;
} else {
    b[::] = 0;
}
```

Scalar function call:

```c
void f (double v) {
    printf( "%lf \n", v );
} f(a[::]);
```
Advanced example expressions

- Conditions:

```c
if (a[::] == 1) {
    b[::] = 42;
} else {
    b[::] = 0;
}
```

- Scalar function call:

```c
void f(double v) {
    printf("%lf \n", v);
}

f(a[::]);
```
Intel Cilk Plus Arrays

Advanced example expressions (cont’d)

- Gather:

```c
1 c[::] = a[b[::]];
```
Advanced example expressions (cont’d)

- Gather:
  
  ```
  c[:] = a[b[:]];  
  ```

- Scatter:
  
  ```
  a[b[:]] = c[:];  
  ```
Intel Cilk Plus Arrays

Advanced example expressions (cont’d)

- **Gather:**
  
  
  ```
  c[:] = a[b[:]];  
  ```

- **Scatter:**
  
  ```
  a[b[:]] = c[:];  
  ```

- **Reductions:**
  
  ```
  __sec_reduce_add (a[:]);  
  __sec_reduce_mul (a[:]);  
  ```

  ```
  . . .  
  ```
Other Cilk Plus Ports

Cilk Plus / GCC
- Integrated in GCC 4.9.2
  - Tasks
  - Array notation
  - No cilk_for keyword yet

Usage

```bash
1  g++ -fcilkplus -lcilkrt -o fib fib.cpp
```

Cilk Plus / LLVM
- URL: http://cilkplus.github.io/
Examples using Cilk Plus

- Tasks
- Arrays
Conclusion
Conclusion

Cilk: A Forerunning Language for Parallel Programming

From the programmer point of view

- Easy to program with (only 3-4 keywords)
- Efficient on *Divide and Conquer* application classes

From the HPC research point of view

- Key precursory effort in promoting task parallelism
- Inspirational for many related works over the last 20 years

*The Implementation of the Cilk-5 Multithreaded Language*
Related Works
Variants and Evolutions of the Task Scheduling Model

Some related works Inspired by Cilk

- OpenMP 3.0 tasks
- Intel Threading Building Blocks (TBB)
Variants and Evolutions of the Task Scheduling Model

Some related works Inspired by Cilk
- OpenMP 3.0 tasks
- Intel Threading Building Blocks (TBB)

Evolutions of the task model
- Dependent Tasks
- Heterogeneous Tasks
Dependent Tasks

More flexibility in specifying task relationships

- Various possibilities
  - Task – Task dependencies
  - Task – Data dependencies

- Various expressiveness vs automatism trade-offs
  - Expressed dependencies
  - Inferred dependencies
Dependent Tasks

More flexibility in specifying task relationships
- Various possibilities
  - Task – Task dependencies
  - Task – Data dependencies
- Various expressiveness vs automatism trade-offs
  - Expressed dependencies
  - Inferred dependencies

Some examples
- StarPU (STORM Team)
- OpenMP 4.0 tasks
- OmpSs (Barcelona Supercomputing Center)
- Kaapi (Inria Team MOAIS)
- DAGuE / PARSeC (UTK + Inria Team HiePACS)
- etc.
Ex.: Sequential Cholesky Decomposition

```c
for (j = 0; j < N; j++) {
    POTRF ( A[j][j]);
    for (i = j+1; i < N; i++)
        TRSM ( A[i][j], A[j][j]);
    for (i = j+1; i < N; i++) {
        SYRK ( A[i][i], A[i][j]);
        for (k = j+1; k < i; k++)
            GEMM ( A[i][k],
                  A[i][j], A[k][j]);
    }
}
```
Ex.: **Task-Based** Cholesky Decomposition

```c
for (j = 0; j < N; j++) {
    POTRF (RW, A[j][j]);
    for (i = j+1; i < N; i++)
        TRSM (RW, A[i][j], R, A[j][j]);
    for (i = j+1; i < N; i++) {
        SYRK (RW, A[i][i], R, A[i][j]);
        for (k = j+1; k < i; k++)
            GEMM (RW, A[i][k],
                 R, A[i][j], R, A[k][j]);
    }
}
__wait__();
```
Ex.: Inferred Dependencies

```c
for (j = 0; j < N; j++) {
    POTRF (RW,A[j][j]);
    for (i = j+1; i < N; i++)
        TRSM (RW,A[i][j], R,A[j][j]);
    for (i = j+1; i < N; i++) {
        SYRK (RW,A[i][i], R,A[i][j]);
        for (k = j+1; k < i; k++)
            GEMM (RW,A[i][k],
                  R,A[i][j], R,A[k][j]);
    }
}
__wait__();
```
Ex.: Inferred Dependencies

```c
for (j = 0; j < N; j++) {
  POTRF (RW, A[j][j]);
  for (i = j+1; i < N; i++)
    TRSM (RW, A[i][j], R, A[j][j]);
  for (i = j+1; i < N; i++) {
    SYRK (RW, A[i][i], R, A[i][j]);
    for (k = j+1; k < i; k++)
      GEMM (RW, A[i][k],
           R, A[i][j], R, A[k][j]);
  }
}
__wait__();
```
for (j = 0; j < N; j++) {
    POTRF (RW,A[j][j]);
    for (i = j+1; i < N; i++)
        TRSM (RW,A[i][j], R,A[j][j]);
    for (i = j+1; i < N; i++) {
        SYRK (RW,A[i][i], R,A[i][j]);
        for (k = j+1; k < i; k++)
            GEMM (RW,A[i][k],
                   R,A[i][j], R,A[k][j]);
    }
}__wait__();
Ex.: Inferred Dependencies

```c
for (j = 0; j < N; j++) {
    POTRF (RW,A[j][j]);
    for (i = j+1; i < N; i++)
        TRSM (RW,A[i][j], R,A[j][j]);
    for (i = j+1; i < N; i++) {
        SYRK (RW,A[i][i], R,A[i][j]);
        for (k = j+1; k < i; k++)
            GEMM (RW,A[i][k],
                  R,A[i][j], R,A[k][j]);
    }
}
__wait__();
```
Ex.: Inferred Dependencies

```c
for (j = 0; j < N; j++) {
    POTRF (RW,A[j][j]);
    for (i = j+1; i < N; i++)
        TRSM (RW,A[i][j], R,A[j][j]);
    for (i = j+1; i < N; i++) {
        SYRK (RW,A[i][i], R,A[i][j]);
        for (k = j+1; k < i; k++)
            GEMM (RW,A[i][k],
                R,A[i][j], R,A[k][j]);
    }
}
__wait__();
```
Ex.: Inferred Dependencies

```c
for (j = 0; j < N; j++) {
    POTRF (RW, A[j][j]);
    for (i = j+1; i < N; i++)
        TRSM (RW, A[i][j], R, A[j][j]);
    for (i = j+1; i < N; i++) {
        SYRK (RW, A[i][i], R, A[i][j]);
        for (k = j+1; k < i; k++)
            GEMM (RW, A[i][k],
                  R, A[i][j], R, A[k][j]);
    }
}
__wait__();
```
for (j = 0; j < N; j++) {
    POTRF (RW, A[j][j]);
    for (i = j+1; i < N; i++)
        TRSM (RW, A[i][j], R, A[j][j]);
    for (i = j+1; i < N; i++) {
        SYRK (RW, A[i][i], R, A[i][j]);
        for (k = j+1; k < i; k++)
            GEMM (RW, A[i][k],
                  R, A[i][j], R, A[k][j]);
    }
}
__wait__();
Ex.: Inferred Dependencies

```c
for (j = 0; j < N; j++) {
    POTRF (RW, A[j][j]);
    for (i = j+1; i < N; i++)
        TRSM (RW, A[i][j], R, A[j][j]);
    for (i = j+1; i < N; i++) {
        SYRK (RW, A[i][i], R, A[i][j]);
        for (k = j+1; k < i; k++)
            GEMM (RW, A[i][k], R, A[i][j], R, A[k][j]);
    }
}__wait__();
```
for (j = 0; j < N; j++) {
    POTRF (RW,A[j][j]);
    for (i = j+1; i < N; i++)
        TRSM (RW,A[i][j], R,A[j][j]);
    for (i = j+1; i < N; i++) {
        SYRK (RW,A[i][i], R,A[i][j]);
        for (k = j+1; k < i; k++)
            GEMM (RW,A[i][k],
                  R,A[i][j], R,A[k][j]);
    }
}__wait__();
for (j = 0; j < N; j++) {
    POTRF (RW,A[j][j]);
    for (i = j+1; i < N; i++)
        TRSM (RW,A[i][j], R,A[j][j]);
    for (i = j+1; i < N; i++) {
        SYRK (RW,A[i][i], R,A[i][j]);
        for (k = j+1; k < i; k++)
            GEMM (RW,A[i][k],
                  R,A[i][j], R,A[k][j]);
    }
}
__wait__();
Ex.: Inferred Dependencies

```c
for (j = 0; j < N; j++) {
    POTRF (RW,A[j][j]);
    for (i = j+1; i < N; i++)
        TRSM (RW,A[i][j], R,A[j][j]);
    for (i = j+1; i < N; i++) {
        SYRK (RW,A[i][i], R,A[i][j]);
        for (k = j+1; k < i; k++)
            GEMM (RW,A[i][k],
                 R,A[i][j], R,A[k][j]);
    }
}
__wait__();
```
Ex.: Inferred Dependencies

```c
for (j = 0; j < N; j++) {
    POTRF (RW, A[j][j]);
    for (i = j+1; i < N; i++)
        TRSM (RW, A[i][j], R, A[j][j]);
    for (i = j+1; i < N; i++) {
        SYRK (RW, A[i][i], R, A[i][j]);
        for (k = j+1; k < i; k++)
            GEMM (RW, A[i][k],
                  R, A[i][j], R, A[k][j]);
    }
}
__wait__();
```
Ex.: Inferred Dependencies

for (j = 0; j < N; j++) {
    POTRF (RW,A[j][j]);
    for (i = j+1; i < N; i++)
        TRSM (RW,A[i][j], R,A[j][j]);
    for (i = j+1; i < N; i++) {
        SYRK (RW,A[i][i], R,A[i][j]);
        for (k = j+1; k < i; k++)
            GEMM (RW,A[i][k],
                  R,A[i][j], R,A[k][j]);
    }
}
__wait__();
Heterogeneous Tasks

Special purpose accelerator devices (or general purpose GPUs)

- (initially) a discrete expansion card
- Rationale: dye area trade-off
Heterogeneous Tasks

Special purpose accelerator devices (or general purpose GPUs)
- (initially) a discrete expansion card
- Rationale: dye area trade-off

Single Instruction Multiple Threads (SIMT)
- A single control unit...
- ... for several computing units
Heterogeneous Tasks

Special purpose accelerator devices (or general purpose GPUs)
- (initially) a discrete expansion card
- Rationale: dye area trade-off

Single Instruction Multiple Threads (SIMT)
- A single control unit . . .
- . . . for several computing units
Heterogeneous Tasks

Special purpose accelerator devices (or general purpose GPUs)

- (initially) a discrete expansion card
- Rationale: dye area trade-off

Single Instruction Multiple Threads (SIMT)

- A single control unit...
- ... for several computing units
- Allows flows to diverge
- ... but better avoid it!

```
if(cond){
  ...
  ...
  ...
} else {
  ...
  ...
}
```
GPU Hardware Model

CPU vs GPU

- **Multiple strategies for multiple purposes**

  **CPU**
  - Strategy
    - Large caches
    - Large control
  - Purpose
    - Complex codes, branching
    - Complex memory access patterns
  - World Rally Championship car

  **GPU**
  - Strategy
    - Lot of computing power
    - Simplified control
  - Purpose
    - Regular data parallel codes
    - Simple memory access patterns
  - Formula One car
Heterogeneous Tasks

Main issues

- Offload or not offload?
  - Computation kernel efficiency on accelerator wrt on the main CPU?
- Discrete accelerator memory
  - Data transfer management
  - Data transfer cost

Some examples

- StarPU (STORM Team)
- OmpSs (Barcelona Supercomputing Center)
- DAGuE / PARSeC (UTK + Inria Team HiePACS)
StarPU Showcase with the MAGMA Linear Algebra Library

UTK, INRIA HIEPACS, INRIA RUNTIME

- QR decomposition on 16 CPUs (AMD) + 4 GPUs (C1060)

![Graph showing performance of different configurations of CPUs and GPUs]

**Measured increase:**
- +12 CPUs
- ~200 GFlops

**Expected increase:**
- +12 CPUs
- ~150 Gflops
Showcase with the MAGMA Linear Algebra Library

**QR kernel properties**

<table>
<thead>
<tr>
<th>Kernel</th>
<th>CPU:</th>
<th>GPU:</th>
<th>Speed-up:</th>
</tr>
</thead>
<tbody>
<tr>
<td>SGEQRT</td>
<td>9 GFlop/s</td>
<td>30 GFlop/s</td>
<td>3</td>
</tr>
<tr>
<td>STSQRT</td>
<td>12 GFlop/s</td>
<td>37 GFlop/s</td>
<td>3</td>
</tr>
<tr>
<td>SOMQRT</td>
<td>8.5 GFlop/s</td>
<td>227 GFlop/s</td>
<td>27</td>
</tr>
<tr>
<td>SSSMQ</td>
<td>10 GFlop/s</td>
<td>285 GFlop/s</td>
<td>28</td>
</tr>
</tbody>
</table>

**Consequences**

- Task distribution
  - SGEQRT: 20% Tasks on GPU
  - SSSMQ: 92% tasks on GPU
- **StarPU takes advantage of heterogeneity!**
  - Only use the accelerator for kernels it is good for
  - Don’t slow down the accelerator with kernels it is not good for
Upcoming Tutorial

PRACE PATC Training session on Runtime Systems

- At INRIA in Bordeaux, France
- June 4-5, 2015
- In partnership with La Maison de la Simulation
- http://www.maisondelasimulation.fr/

Program

- Hwloc: locality management tool/lib
- StarPU: task scheduler
- EZtrace: trace-based debugging framework
- Kaapi: task scheduler