LECTURE 5
Multicore Programming

Saman Amarasinghe
2014
Project 1 Beta – Performance

Project 1 beta: geometric mean time (s)

Username (omitted)
Transistor count is still rising, ..., but clock speed is bounded at ~4GHz.
Power density, had scaling of clock frequency continued its trend of 25%–30% increase per year.

Technology Scaling

Each generation of Moore’s Law potentially doubles the number of cores.
Abstract Multicore Architecture

Chip Multiprocessor (CMP)
OUTLINE

• Shared-Memory Hardware
• Concurrency Platforms
  ▪ Pthreads (and WinAPI Threads)
  ▪ Threading Building Blocks
  ▪ OpenMP
  ▪ Cilk Plus

© 2013 Charles E. Leiserson and Saman P. Amarasinghe
Cache Coherence

Load $x$

$\ldots$

© 2013 Charles E. Leiserson and Saman P. Amarasinghe
Cache Coherence

Load x
Cache Coherence

Load $x$

$x = 3$

P

...
Cache Coherence

Load x
Cache Coherence

Store x
Cache Coherence

Load $x$

$P$ $x=3$ $P$ $x=3$ $P$ $x=5$

...
Each cache line is labeled with a state:

- **M**: cache block has been modified. No other caches contain this block in M or S states. Different from main memory.

- **E**: No other caches has this block. Same as main memory.

- **S**: other caches may be sharing this block. Same as main memory.

- **I**: cache block is invalid (same as not there).
MESI Protocol

Each cache line is labeled with a state:
• **M**: cache block has been modified. No other caches contain this block. Different from main memory
• **E**: No other caches has this block. Same as main memory
• **S**: other caches may be sharing this block. Same as main memory
• **I**: cache block is invalid (same as not there).

Read W

```
E: w=13
```

```
w=13  x=17  y=12  z=8
```
Each cache line is labeled with a state:

- **M**: cache block has been modified. No other caches contain this block. Different from main memory
- **E**: No other caches has this block. Same as main memory
- **S**: other caches may be sharing this block. Same as main memory
- **I**: cache block is invalid (same as not there).

**Write X**

- **M**: $x=15$
- **E**: $w=13$
- $w=13$
- $x=17$
- $y=12$
- $z=8$
MESI Protocol

Each cache line is labeled with a state:

- **M**: cache block has been modified. No other caches contain this block. Different from main memory.
- **E**: No other caches has this block. Same as main memory.
- **S**: other caches may be sharing this block. Same as main memory.
- **I**: cache block is invalid (same as not there).

Read Y

- **M**: x=17
- **E**: w=13
- **E**: y=12
- **I**: z=8
MESI Protocol

Each cache line is labeled with a state:

- **M**: cache block has been modified. No other caches contain this block. Different from main memory
- **E**: No other caches has this block. Same as main memory
- **S**: other caches may be sharing this block. Same as main memory
- **I**: cache block is invalid (same as not there).

Read Y

<table>
<thead>
<tr>
<th>M:</th>
<th>x=17</th>
</tr>
</thead>
<tbody>
<tr>
<td>E:</td>
<td>w=13</td>
</tr>
<tr>
<td>S:</td>
<td>y=12</td>
</tr>
<tr>
<td>w=13</td>
<td>x=17</td>
</tr>
</tbody>
</table>
Each cache line is labeled with a state:

- **M:** cache block has been \textit{modified}. No other caches contain this block. Different from main memory
- **E:** No other caches has this block. Same as main memory
- **S:** other caches may be \textit{sharing} this block. Same as main memory
- **I:** cache block is \textit{invalid} (same as not there).

\begin{tikzpicture}
  \node (M) at (0,0) [green, rounded corners, draw] {
    \begin{tabular}{l}
      M: x=17 \\
      S: z=8
    \end{tabular}
  };
  \node (E) at (1.5,0) [green, rounded corners, draw] {
    \begin{tabular}{l}
      E: w=13 \\
      S: y=12
    \end{tabular}
  };
  \node (S) at (3,0) [green, rounded corners, draw] {
    \begin{tabular}{l}
      S: z=8
    \end{tabular}
  };
  \node (I) at (4.5,0) [green, rounded corners, draw] {
    \begin{tabular}{l}
      S: y=12
    \end{tabular}
  };
  \node (yellow) at (0,-1) [orange, rounded corners, draw] {
    \begin{tabular}{l}
      w=13 \\
      x=17 \\
      y=12 \\
      z=8
    \end{tabular}
  };
\end{tikzpicture}
OUTLINE

• Shared-Memory Hardware

• Concurrency Platforms
  ▪ Pthreads (and WinAPI Threads)
  ▪ Threading Building Blocks
  ▪ OpenMP
  ▪ Cilk Plus
Concurrency Platforms

- Programming directly on processor cores is painful and error-prone.
- A **concurrency platform** abstracts processor cores, handles synchronization and communication protocols, and performs load balancing.
- **Examples**
  - Pthreads and WinAPI threads
  - Threading Building Blocks (TBB)
  - OpenMP
  - Cilk Plus
Fibonacci Numbers

The Fibonacci numbers are the sequence \( \langle 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots \rangle \), where each number is the sum of the previous two.

Recurrence:
- \( F_0 = 0 \),
- \( F_1 = 1 \),
- \( F_n = F_{n-1} + F_{n-2} \) for \( n > 1 \).

The sequence is named after Leonardo di Pisa (1170–1250 A.D.), also known as Fibonacci, a contraction of *filius Bonaccii* —“son of Bonaccio.” Fibonacci’s 1202 book *Liber Abaci* introduced the sequence to Western mathematics, although it had previously been discovered by Indian mathematicians.
```c
#include <inttypes.h>
#include <stdio.h>
#include <stdlib.h>

uint64_t fib(uint64_t n) {
    if (n < 2) {
        return n;
    } else {
        uint64_t x = fib(n-1);
        uint64_t y = fib(n-2);
        return (x + y);
    }
}

int main(int argc, char *argv[]) {
    uint64_t n = atoi(argv[1]);
    uint64_t result = fib(n);
    printf("Fibonacci of %" PRIu64 " is %" PRIu64 ".\n", n, result);
    return 0;
}
```

Disclaimer to Algorithms Police
This recursive program is a poor way to compute the $n$th Fibonacci number, but it provides for a good didactic example.
Fibonacci Execution

Key idea for parallelization
The calculations of fib(n-1) and fib(n-2) can be executed simultaneously without mutual interference.

uint64_t fib(uint64_t n) {
    if (n < 2) {
        return n;
    } else {
        uint64_t x = fib(n-1);
        uint64_t y = fib(n-2);
        return (x + y);
    }
}
• Shared-Memory Hardware
• Concurrency Platforms
  ▪ Pthreads (and WinAPI Threads)
  ▪ Threading Building Blocks
  ▪ OpenMP
  ▪ Cilk Plus
Pthreads*

- **Do-it-yourself** concurrency platform.
- Built as a library of functions with “special” non-C semantics.
- Each thread implements an *abstraction of a processor*, which are multiplexed onto machine resources.
- Threads communicate though shared memory.
- Library functions mask the protocols involved in interthread coordination.

---

*WinAPI threads* provide similar functionality.
Key Pthread Functions

```
int pthread_create(
    pthread_t *thread,
    const pthread_attr_t *attr,
    void *(*func)(void *),
    void *arg
)
```
Key Pthread Functions

```c
int pthread_create(
    pthread_t *thread,
    // returned identifier for the new thread
    const pthread_attr_t *attr,
    // object to set thread attributes (NULL for default)
    void *(*func)(void *),
    // routine executed after creation
    void *arg
    // a single argument passed to func
) // returns error status
```

```c
int pthread_join(
    pthread_t thread,
    void **status
)
```
Key Pthread Functions

```c
int pthread_create(
    pthread_t *thread,
    // returned identifier for the new thread
    const pthread_attr_t *attr,
    // object to set thread attributes (NULL for default)
    void **func(void *),
    // routine executed after creation
    void *arg
    // a single argument passed to func
) // returns error status
```

```c
int pthread_join(
    pthread_t thread,
    // identifier of thread to wait for
    void **status
    // terminating thread’s status (NULL to ignore)
) // returns error status
```
#include <inttypes.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

uint64_t fib(uint64_t n) {
    if (n < 2) {
        return n;
    } else {
        uint64_t x = fib(n-1);
        uint64_t y = fib(n-2);
        return (x + y);
    }
}
#include <inttypes.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

uint64_t fib(uint64_t n) {
    if (n < 2) {
        return n;
    } else {
        uint64_t x = fib(n-1);
        uint64_t y = fib(n-2);
        return (x + y);
    }
}

typedef struct {
    uint64_t input;
    uint64_t output;
} thread_args;

Pthread Implementation Structure for thread arguments.
#include <inttypes.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

uint64_t fib(uint64_t n) {
    if (n < 2) {
        return n;
    } else {
        uint64_t x = fib(n-1);
        uint64_t y = fib(n-2);
        return (x + y);
    }
}

typedef struct {
    uint64_t input;
    uint64_t output;
} thread_args;

void *thread_func(void *ptr) {
    uint64_t i = ((thread_args *) ptr)->input;
    ((thread_args *) ptr)->output = fib(i);
    return NULL;
}
#include <inttypes.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

uint64_t fib(uint64_t n) {
    if (n < 2) {
        return n;
    } else {
        uint64_t x = fib(n-1);
        uint64_t y = fib(n-2);
        return (x + y);
    }
}

typedef struct {
    uint64_t input;
    uint64_t output;
} thread_args;

void *thread_func(void *ptr) {
    uint64_t i =
        ((thread_args *) ptr)->input;
    ((thread_args *) ptr)->output = fib(i);
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t thread;
    thread_args args;
    int status;
    uint64_t result;

    if (argc < 2) { return 1; }
    uint64_t n = strtol(argv[1], NULL, 0);

    printf("Fibonacci of %" PRIu64 " is %" PRIu64 ".\n", n, result);
    return 0;
}
Pthread Implementation

```c
#include <inttypes.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

uint64_t fib(uint64_t n) {
    if (n < 2) {
        return n;
    } else {
        uint64_t x = fib(n-1);
        uint64_t y = fib(n-2);
        return (x + y);
    }
}

typedef struct {
    uint64_t input;
    uint64_t output;
} thread_args;

void *thread_func(void *ptr) {
    uint64_t i = ((thread_args *) ptr)->input;
    ((thread_args *) ptr)->output = fib(i);
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t thread;
    thread_args args;
    int status;
    uint64_t result;

    if (argc < 2) { return 1; }
    uint64_t n = strtoul(argv[1], NULL, 0);
    if (n < 30) {
        result = fib(n);
    } else {

        No point in creating thread if there isn’t enough to do.

    }
    printf("Fibonacci of %" PRIu64 " is %" PRIu64 ".\n", n, result);
    return 0;
}
```

© 2013 Charles E. Leiserson and Saman P. Amarasinghe
```c
#include <inttypes.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

uint64_t fib(uint64_t n) {
    if (n < 2) {
        return n;
    } else {
        uint64_t x = fib(n-1);
        uint64_t y = fib(n-2);
        return (x + y);
    }
}

typedef struct {
    uint64_t input;
    uint64_t output;
} thread_args;

void *thread_func(void *ptr) {
    uint64_t i =
    ((thread_args *) ptr)->input;
    ((thread_args *) ptr)->output = fib(i);
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t thread;
    thread_args args;
    int status;
    uint64_t result;
    if (argc < 2) { return 1; }
    uint64_t n = strtoul(argv[1], NULL, 0);
    if (n < 30) {
        result = fib(n);
    } else {
        args.input = n-1;
    }
    printf("Fibonacci of %" PRIu64 " is %" PRIu64 "\n",
           n, result);
    return 0;
}
```

Marshal input argument to thread.
#include <inttypes.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

uint64_t fib(uint64_t n) {
    if (n < 2) {
        return n;
    } else {
        uint64_t x = fib(n-1);
        uint64_t y = fib(n-2);
        return (x + y);
    }
}

typedef struct {
    uint64_t input;
    uint64_t output;
} thread_args;

void *thread_func(void *ptr) {
    uint64_t i = ((thread_args *) ptr)->input;
    ((thread_args *) ptr)->output = fib(i);
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t thread;
    thread_args args;
    int status;
    uint64_t result;

    if (argc < 2) { return 1; }
    uint64_t n = strtoul(argv[1], NULL, 0);
    if (n < 30) {
        result = fib(n);
    } else {
        args.input = n-1;
        status = pthread_create(&thread, NULL, thread_func, (void*) &args);
    }

    printf("Fibonacci of %" PRIu64 " is %" PRIu64 ".\n", n, result);
    return 0;
}
#include <inttypes.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

uint64_t fib(uint64_t n) {
    if (n < 2) {
        return n;
    } else {
        uint64_t x = fib(n-1);
        uint64_t y = fib(n-2);
        return (x + y);
    }
}

typedef struct {
    uint64_t input;
    uint64_t output;
} thread_args;

void *thread_func(void *ptr) {
    uint64_t i = ((thread_args *) ptr)->input;
    ((thread_args *) ptr)->output = fib(i);
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t thread;
    thread_args args;
    int status;
    uint64_t result;

    if (argc < 2) { return 1; }
    uint64_t n = strtoul(argv[1], NULL, 0);
    if (n < 30) {
        result = fib(n);
    } else {
        args.input = n-1;
        status = pthread_create(&thread, NULL, thread_func, (void*) &args);
        // main can continue executing
        if (status != NULL) { return 1; }
        result = fib(n-2);
    }

    printf("Fibonacci of %" PRIu64 " is %" PRIu64 "\n", n, result);
    return 0;
}
#include <inttypes.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

uint64_t fib(uint64_t n) {
    if (n < 2) {
        return n;
    } else {
        uint64_t x = fib(n-1);
        uint64_t y = fib(n-2);
        return (x + y);
    }
}

typedef struct {
    uint64_t input;
    uint64_t output;
} thread_args;

void *thread_func(void *ptr) {
    uint64_t i = ((thread_args *) ptr)->input;
    ((thread_args *) ptr)->output = fib(i);
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t thread;
    thread_args args;
    int status;
    uint64_t result;
    
    if (argc < 2) { return 1; }
    uint64_t n = strtoul(argv[1], NULL, 0);
    if (n < 30) {
        result = fib(n);
    } else {
        args.input = n-1;
        status = pthread_create(&thread, NULL, thread_func, (void*) &args);
        // main can continue executing
        if (status != NULL) { return 1; }
        result = fib(n-2);
        // Wait for the thread to terminate.
        status = pthread_join(thread, NULL);
        if (status != NULL) { return 1; }
    }
    printf("Fibonacci of %" PRIu64 " is %" PRIu64 "\n", n, result);
    return 0;
}
#include <inttypes.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

uint64_t fib(uint64_t n) {
    if (n < 2) {
        return n;
    } else {
        uint64_t x = fib(n-1);
        uint64_t y = fib(n-2);
        return (x + y);
    }
}

typedef struct {
    uint64_t input;
    uint64_t output;
} thread_args;

void *thread_func(void *ptr) {
    uint64_t i =
        ((thread_args *) ptr)->input;
    ((thread_args *) ptr)->output = fib(i);
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t thread;
    thread_args args;
    int status;
    uint64_t result;

    if (argc < 2) { return 1; }
    uint64_t n = strtoul(argv[1], NULL, 0);
    if (n < 30) {
        result = fib(n);
    } else {
        args.input = n-1;
        status = pthread_create(&thread, NULL,
            thread_func, (void*) &args);

        // main can continue executing
        if (status != NULL) { return 1; }
        result = fib(n-2);

        // Wait for the thread to terminate.
        status = pthread_join(thread, NULL);
        if (status != NULL) { return 1; }
        result += args.output;
    }
    printf("Fibonacci of %" PRIu64 " is %" PRIu64 ".\n", n, result);
    return 0;
}
## Issues with Pthreads

| Overhead | The cost of creating a thread $>10^4$ cycles ⇒ coarse-grained concurrency. (Thread pools can help.) |
### Issues with Pthreads

<table>
<thead>
<tr>
<th>Overhead</th>
<th>The cost of creating a thread $&gt;10^4$ cycles ⇒ coarse-grained concurrency. (Thread pools can help.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalability</td>
<td>Fibonacci code gets at most about 1.5 speedup for 2 cores. Need a rewrite for more cores.</td>
</tr>
</tbody>
</table>
## Issues with Pthreads

<table>
<thead>
<tr>
<th>Issue</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overhead</td>
<td>The cost of creating a thread ( &gt;10^4 ) cycles ⇒ coarse-grained concurrency. (Thread pools can help.)</td>
</tr>
<tr>
<td>Scalability</td>
<td>Fibonacci code gets at most about 1.5 speedup for 2 cores. Need a rewrite for more cores.</td>
</tr>
<tr>
<td>Modularity</td>
<td>The Fibonacci logic is no longer neatly encapsulated in the \texttt{fib()} function.</td>
</tr>
</tbody>
</table>
## Issues with Pthreads

<table>
<thead>
<tr>
<th>Overhead</th>
<th>The cost of creating a thread $&gt;10^4$ cycles ⇒ coarse–grained concurrency. (Thread pools can help.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalability</td>
<td>Fibonacci code gets at most about 1.5 speedup for 2 cores. Need a rewrite for more cores.</td>
</tr>
<tr>
<td>Modularity</td>
<td>The Fibonacci logic is no longer neatly encapsulated in the <code>fib()</code> function.</td>
</tr>
</tbody>
</table>
• Shared-Memory Hardware
• Concurrency Platforms
  ▪ Pthreads (and WinAPI Threads)
  ▪ Threading Building Blocks
  ▪ OpenMP
  ▪ Cilk Plus
Threading Building Blocks

- Developed by Intel.
-Implemented as a C++ library that runs on top of native threads.
- Programmer specifies tasks rather than threads.
- Tasks are automatically load balanced across the threads using work-stealing.
- Focus on performance.
using namespace tbb;
class FibTask: public task {
public:
const uint64_t n;
uint64_t* const sum;
FibTask(uint64_t n_, uint64_t* sum_):
n(n_), sum(sum_) {} 

task* execute() {
if( n < 2 ) {
    *sum = n;
} else {
    uint64_t x, y;
    FibTask& a = *new( allocate_child() )
        FibTask(n-1, &x);
    FibTask& b = *new( allocate_child() )
        FibTask(n-2, &y);
    set_ref_count(3);
    spawn(b);
    spawn_and_wait_for_all(a);
    *sum = x + y;
}
    return NULL;
}
};

#include <cstdint>
#include <iostream>
#include "tbb/task.h"

int main(int argc, char *argv[]) {
    uint64_t res;
    if (argc < 2) { return 1; }
    uint64_t n = 
        strtoul(argv[1], null, 0);
    FibTask& a = *new(task::allocate_root())
        FibTask(n, &res);
    task::spawn_root_and_wait(a);
    std::cout << "Fibonacci of " << n
        << " is " << res << std::endl;
    return 0;
}
using namespace tbb;
class FibTask: public task {
public:
    const uint64_t n;
    uint64_t* const sum;
    FibTask(uint64_t n_, uint64_t* sum_): 
n(n_), sum(sum_) {} 

    task* execute() {
        if( n < 2 ) {
            *sum = n;
        } else {
            uint64_t x, y;
            FibTask& a = *new( allocate_child() ) 
                FibTask(n-1, &x);
            FibTask& b = *new( allocate_child() )
                FibTask(n-2, &y);
            set_ref_count(3);
            spawn(b);
            spawn_and_wait_for_all(a);
            *sum = x + y;
        }
        return NULL;
    }
};

#include <cstdint>
#include <iostream>
#include "tbb/task.h"

int main(int argc, char *argv[]) {
    uint64_t res;
    if (argc < 2) { return 1; }
    uint64_t n = 
        strtoul(argv[1], NULL, 0);
    FibTask& a = *new(task::allocate_root()) 
        FibTask(n, &res);
    task::spawn_root_and_wait(a);
}
    std::cout << "Fibonacci of " << n 
        " is " << res << std::endl;
    return 0;
}
using namespace tbb;
class FibTask: public task {
public:
    const uint64_t n;
    uint64_t* const sum;
    FibTask(uint64_t n_, uint64_t* sum_):
        n(n_), sum(sum_) {}

task* execute() {
    if (n < 2) {
        *sum = n;
    } else {
        uint64_t x, y;
        FibTask& a = *new( allocate_child() )
            FibTask(n-1, &x);
        FibTask& b = *new( allocate_child() )
            FibTask(n-2, &y);
        set_ref_count(3);
        spawn(b);
        spawn_and_wait_for_all(a);
        *sum = x + y;
    }
    return NULL;
}
};

FibTask has an input parameter n and an output parameter sum.

#include <cstdint>
#include <iostream>
#include "tbb/task.h"

int main(int argc, char *argv[]) {
    uint64_t res;
    if (argc < 2) { return 1; }
    uint64_t n = strtoul(argv[1], NULL, 0);
    FibTask& a = *new(task::allocate_root())
        FibTask(n, &res);
    task::spawn_root_and_wait(a);
    std::cout << "Fibonacci of " << n
               << " is " << res << std::endl;
    return 0;
}
Fibonacci in TBB

```
using namespace tbb;
class FibTask: public task {
public:
    const uint64_t n;
    uint64_t* const sum;
    FibTask(uint64_t n_, uint64_t* sum_)
        : n(n_), sum(sum_) {}  

    task* execute() {
        if (n < 2) {
            *sum = n;
        } else {
            uint64_t x, y;
            FibTask& a = *new( allocate_child() )
                FibTask(n-1, &x);
            FibTask& b = *new( allocate_child() )
                FibTask(n-2, &y);
            set_ref_count(3);
            spawn(b);
            spawn_and_wait_for_all(a);
            *sum = x + y;
        }
        return NULL;
    }
};
```

The `execute()` function performs the computation when the task is started.

```
#include <cstdint>
#include <iostream>
#include "tbb/task.h"

int main(int argc, char *argv[]) {
    uint64_t res;
    if (argc < 2) { return 1; }
    uint64_t n = strtoul(argv[1], NULL, 0);
    FibTask& a = *new(task::allocate_root())
        FibTask(n, &res);
    task::spawn_root_and_wait(a);
    std::cout << "Fibonacci of " << n
        << " is " << res << std::endl;
    return 0;
}
```
using namespace tbb;
class FibTask: public task {
public:
    const uint64_t n;
    uint64_t* const sum;
    FibTask(uint64_t n_, uint64_t* sum_):
        n(n_), sum(sum_) {}

task* execute() {
    if( n < 2 ) {
        *sum = n;
    } else {
        uint64_t x, y;
        FibTask& a = *new( allocate_child() )
            FibTask(n-1, &x);
        FibTask& b = *new( allocate_child() )
            FibTask(n-2, &y);
        set_ref_count(3);
        spawn(b);
        spawn_and_wait_for_all(a);
        *sum = x + y;
    }
    return NULL;
}
Fibonacci in TBB

```cpp
#include <cstdint>
#include <iostream>
#include "tbb/task.h"

int main(int argc, char *argv[]) {
    uint64_t res;
    if (argc < 2) { return 1; }
    uint64_t n = strtoul(argv[1], NULL, 0);
    FibTask& a = *new(task::allocate_root())
        FibTask(n-1, &x);
    FibTask& b = *new(task::allocate_child())
        FibTask(n-2, &y);
    set_ref_count(3);
    spawn(b);
    spawn_and_wait_for_all(a);
    *sum = x + y;
}
return NULL;
}
```

Set the number of tasks to wait for (2 children + 1 implicit for bookkeeping).
Fibonacci in TBB

using namespace tbb;

class FibTask: public task {
public:
    const uint64_t n;
    uint64_t* const sum;
    FibTask(uint64_t n_, uint64_t* sum_): 
        n(n_), sum(sum_) {}

task* execute() {
    if( n < 2 ) {
        *sum = n;
    } else {
        uint64_t x, y;
        FibTask& a = *new( allocate_child() )
                FibTask(n-1, &x);
        FibTask& b = *new( allocate_child() )
                FibTask(n-2, &y);
        set_ref_count(3);
        spawn(b);
        spawn_and_wait_for_all(a);
        *sum = x + y;
    }
    return NULL;
}
};

#include <cstdint>
#include <iostream>
#include "tbb/task.h"

int main(int argc, char *argv[]) {
    uint64_t res;
    if (argc < 2) { return 1; }
    uint64_t n =
            strtoul(argv[1], NULL, 0);
    FibTask& a = *new(task::allocate_root())
            FibTask(n, &res);
    task::spawn_root_and_wait(a);

    std::cout << "Fibonacci of " << n
        " is " << res << std::endl;
    return 0;
}
#include <cstdint>
#include <iostream>
#include "tbb/task.h"

int main(int argc, char *argv[]) {
    uint64_t res;
    if (argc < 2) { return 1; }
    uint64_t n = strtoul(argv[1], NULL, 0);
    FibTask a = *new(task::allocate_root())
        FibTask(n-1, &x);
    FibTask b = *new(task::allocate_root())
        FibTask(n-2, &y);
    set_ref_count(3);
    spawn(b);
    spawn_and_wait_for_all(a);
    *sum = x + y;
    return NULL;
}

Fibonacci task a and wait for both a and b to finish.
Fibonacci in TBB

```cpp
#include <cstdint>
#include <iostream>
#include "tbb/task.h"

int main(int argc, char *argv[]) {
    uint64_t res;
    if (argc < 2) { return 1; }
    uint64_t n = strtoul(argv[1], NULL, 0);
    FibTask& a = *new(task::allocate_root()) FibTask(n-1, &x);
    FibTask& b = *new(task::allocate_child()) FibTask(n-2, &y);
    set_ref_count(3);
    spawn(b);
    spawn_and_wait_for_all(a);
    *sum = x + y;
    return NULL;
}
```

Add the results together to produce the final output.
Fibonacci in TBB

#include <cstdint>
#include <iostream>
#include "tbb/task.h"

int main(int argc, char *argv[]) {
    uint64_t res;
    if (argc < 2) { return 1; }
    uint64_t n = strtoul(argv[1], NULL, 0);
    FibTask& a = *new(task::allocate_root()) FibTask(n, &res);
    task::spawn_root_and_wait(a);

    std::cout << "Fibonacci of " << n << " is " << res << std::endl;
    return 0;
}

using namespace tbb;
class FibTask: public task {
public:
    const uint64_t n;
    uint64_t* const sum;
    FibTask(uint64_t n_, uint64_t* sum_): n(n_), sum(sum_) {} 

    task* execute() {
        if( n < 2 ) {
            *sum = n;
        } else {
            uint64_t x, y;
            FibTask& a = *new( allocate_child() ) FibTask(n-1, &x);
            FibTask& b = *new( allocate_child() ) FibTask(n-2, &y);
            set_ref_count(3);
            spawn(b);
            spawn_and_wait_for_all(a);
            *sum = x + y;
        }
        return NULL;
    }
};

Create root task; spawn and wait.
Other TBB Features

- **TBB** provides many C++ templates to express common patterns simply, such as
  - `parallel_for` for loop parallelism,
  - `parallel_reduce` for data aggregation,
  - pipeline and filter for software pipelining.
- **TBB** provides **concurrent container** classes, which allow multiple threads to safely access and update items in the container concurrently.
- **TBB** also provides a variety of **mutual-exclusion** library functions, including locks and atomic updates.
OUTLINE

• Shared-Memory Hardware
• Concurrency Platforms
  ▪ Pthreads (and WinAPI Threads)
  ▪ Threading Building Blocks
  ▪ OpenMP
  ▪ Cilk Plus
OpenMP

- Specification produced by an industry consortium.
- Several compilers available, both open-source and proprietary, including GCC, ICC, and Visual Studio.
- Linguistic extensions to C/C++ and Fortran in the form of compiler pragmas.
- Runs on top of native threads.
- Supports loop parallelism and, more recently in Version 3.0, task parallelism.
```c
uint64_t fib(uint64_t n) {
    if (n < 2) {
        return n;
    } else {
        uint64_t x, y;
        #pragma omp task shared(x,n)
        x = fib(n-1);
        #pragma omp task shared(y,n)
        y = fib(n-2);
        #pragma omp taskwait
        return (x + y);
    }
}
```
Fibonacci in OpenMP 3.0

uint64_t fib(uint64_t n) {
    if (n < 2) {
        return n;
    } else {
        uint64_t x, y;
        #pragma omp task shared(x,n)
        x = fib(n-1);
        #pragma omp task shared(y,n)
        y = fib(n-2);
        #pragma omp taskwait
        return (x + y);
    }
}
```c
uint64_t fib(uint64_t n) {
    if (n < 2) {
        return n;
    } else {
        uint64_t x, y;
        #pragma omp task shared(x,n)
        x = fib(n-1);
        #pragma omp task shared(y,n)
        y = fib(n-2);
        #pragma omp taskwait
        return (x + y);
    }
}
```

The following statement is an independent task.
Fibonacci in OpenMP 3.0

```c
uint64_t fib(uint64_t n) {
    if (n < 2) {
        return n;
    } else {
        uint64_t x, y;
        #pragma omp task shared(x,n)
        x = fib(n-1);
        #pragma omp task shared(y,n)
        y = fib(n-2);
        #pragma omp taskwait
        return (x + y);
    }
}
```

Sharing of memory is managed explicitly.
uint64_t fib(uint64_t n) {
    if (n < 2) {
        return n;
    } else {
        uint64_t x, y;
        #pragma omp task shared(x,n)
            x = fib(n-1);
        #pragma omp task shared(y,n)
            y = fib(n-2);
        #pragma omp taskwait
            return (x + y);
    }
}

Wait for the two tasks to complete before continuing.
Other OpenMP Features

- OpenMP provides many 
  **pragma directives** to express common patterns, such as
  - **parallel for** for loop parallelism,
  - **reduction** for data aggregation,
  - directives for scheduling and data sharing.

- OpenMP provides a variety of
  **synchronization constructs**, such as
  barriers, atomic updates, and mutual-exclusion locks.
• Shared–Memory Hardware
• Concurrency Platforms
  ▪ Pthreads (and WinAPI Threads)
  ▪ Threading Building Blocks
  ▪ OpenMP
  ▪ Cilk Plus
The “Cilk” part is a small set of linguistic extensions to C/C++ to support fork–join parallelism. (The “Plus” part supports vector parallelism.)

- Developed originally by Cilk Arts, an MIT spin-off, which was acquired by Intel in July 2009.
- Based on the award–winning Cilk multithreaded language developed at MIT.
- Features a provably efficient work–stealing scheduler.
- Provides a hyperobject library for parallelizing code with global variables.
- Includes the Cilkscreen race detector and Cilkview scalability analyzer.
Nested Parallelism in Cilk

```c
uint64_t fib(uint64_t n) {
    if (n < 2) {
        return n;
    } else {
        uint64_t x, y;
        x = cilk_spawn fib(n-1);
        y = fib(n-2);
        cilk_sync;
        return (x + y);
    }
}
```

The named **child** function may execute in parallel with the **parent** caller.

Control cannot pass this point until all spawned children have returned.

Cilk keywords **grant permission** for parallel execution. They do not **command** parallel execution.
Loop Parallelism in Cilk

Example:
In-place matrix transpose

\[
\begin{pmatrix}
a_{11} & a_{12} & \ldots & a_{1n} \\
a_{21} & a_{22} & \ldots & a_{2n} \\
\vdots & \vdots & \ddots & \vdots \\
a_{n1} & a_{n2} & \ldots & a_{nn}
\end{pmatrix}
\]

\[A\]

\[
\begin{pmatrix}
a_{11} & a_{21} & \ldots & a_{n1} \\
a_{12} & a_{22} & \ldots & a_{n2} \\
\vdots & \vdots & \ddots & \vdots \\
a_{1n} & a_{2n} & \ldots & a_{nn}
\end{pmatrix}
\]

\[A^T\]

The iterations of a cilk_for loop execute in parallel.

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

The **serial elision** of a Cilk program is always a legal interpretation of the program’s semantics.

Remember, Cilk keywords **grant permission** for parallel execution. They do not **command** parallel execution.

To obtain the serial elision:

```c
#define cilk_for for
#define cilk_spawn
#define cilk_sync
```
The Cilk concurrency platform allows the programmer to express logical parallelism in an application.

The Cilk scheduler maps the executing program onto the processor cores dynamically at runtime.

Cilk’s work-stealing scheduler is provably efficient.
The diagram illustrates the Cilk Platform, which includes the following components:

1. **Cilk source**
   - `uint64_t fib(uint64_t n) { if (n < 2) { return n; } else { uint64_t x, y; x = cilk_spawn fib(n-1); y = fib(n-2); cilk_sync; return (x + y); } }

2. **Compiler**

3. **Hyperobject Library**

4. **Runtime System**
   - **Parallel Performance**
   - **Serial elision**

5. **Cilkscreen Race Detector**

6. **Cilkview Scalability Analyzer**

The diagram also shows the conversion of serial elision to parallel code, indicating the use of the `cilk_spawn` function for spawning tasks and `cilk_sync` for synchronization between tasks.

---

© 2013 Charles E. Leiserson and Saman P. Amarasinghe