CS 610: Intel Threading Building Blocks

Swarnendu Biswas

Semester 2020-2021-I

CSE, IIT Kanpur

Copyright Information

- "The instructor of this course owns the copyright of all the course materials. This lecture material was distributed only to the students attending the course CS 610: Programming for Performance of IIT Kanpur, and should not be distributed in print or through electronic media without the consent of the instructor. Students can make their own copies of the course materials for their use."

https://www.iitk.ac.in/doaa/data/FAQ-2020-21-I.pdf
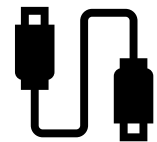
Parallel Programming Overview

Find parallelization opportunities in the problem
- Decompose the problem into parallel units

Create parallel units of execution
- Manage efficient execution of the parallel units

Problem may require inter-unit communication
- Communication between threads, cores, …

Swarnendu Biswas

How to "Think Parallel"?

- Decomposition
  - Decompose the problem into concurrent logical tasks
- Scaling
  - Identify concurrent tasks to keep processors busy
- Choose and utilize appropriate algorithms
- Threads
  - Map tasks to threads
- Correctness
  - Ensure correct synchronization to shared resources

- How much parallelism is there in an application?
  - Depends on the size of the problem
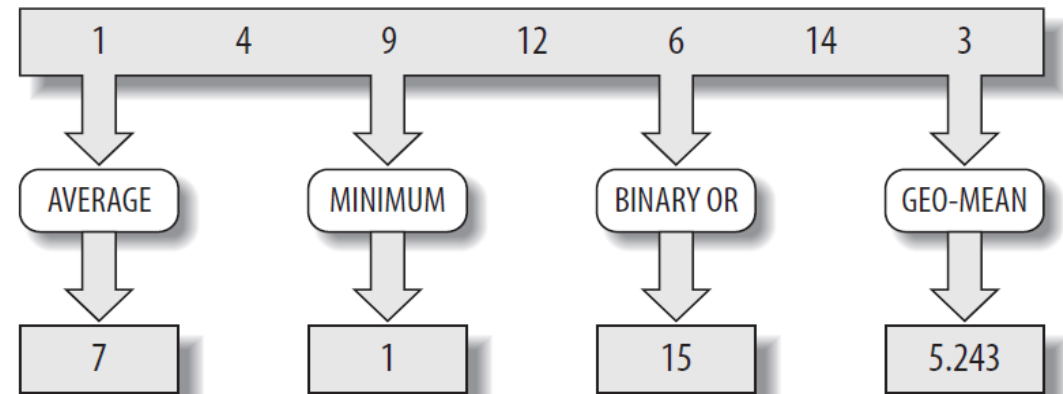 - Depends on whether the algorithm is easily parallelizable

Swarnendu Biswas

How to Decompose?

Data parallelism	Task parallelism

Swarnendu Biswas
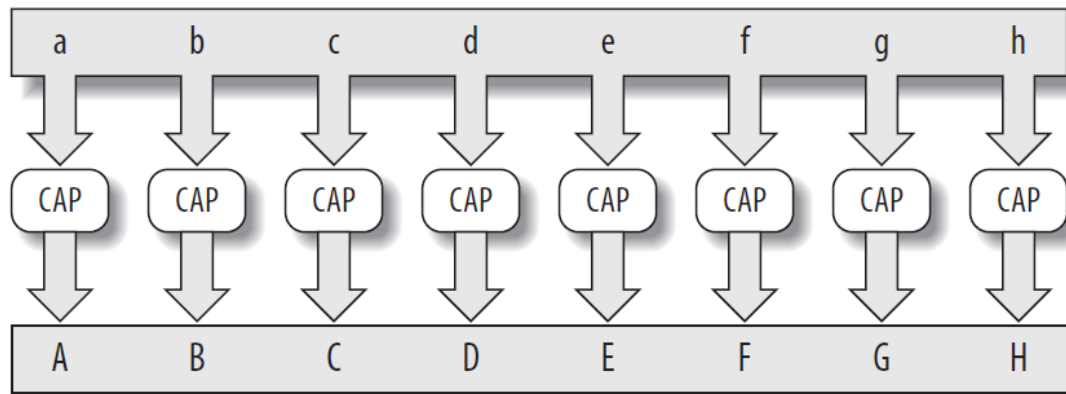
Data Parallelism vs Task Parallelism

| Data Parallelism	Task parallelism
• Same operations performed on different subsets of same data	• Different operations are performed on the same or different data
• Synchronous computation	• Asynchronous computation
• Expected speedup is more as there is only one execution thread operating on all sets of data	• Expected speedup is less as each processor will execute a different thread or process
• Amount of parallelization is proportional to the input data size	• Amount of parallelization is proportional to the number of independent tasks
• Designed for optimum load balance	• Load balancing depends on the availability of the hardware and scheduling algorithms like static and dynamic scheduling

Data Parallelism vs Task Parallelism

- Distinguishing just between data and task parallelism may not be perfect
  - Imagine TAs grading questions of varied difficulty
- Might need hybrid parallelism or pipelining or work stealing

Parallelism vs Concurrency
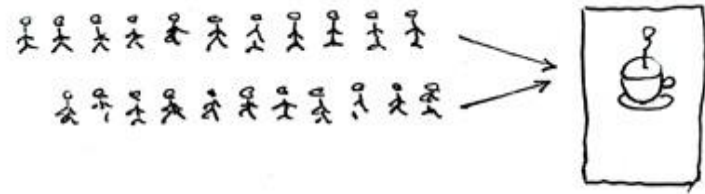
Swarnendu Biswas
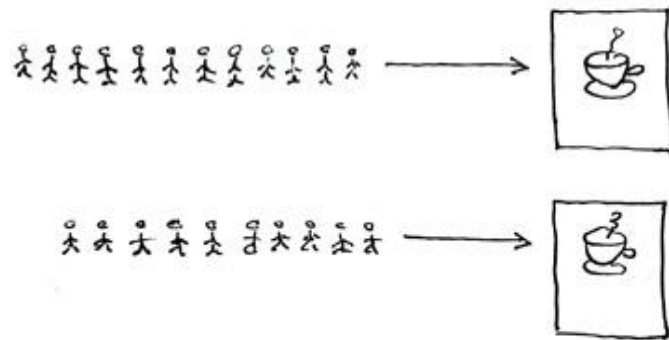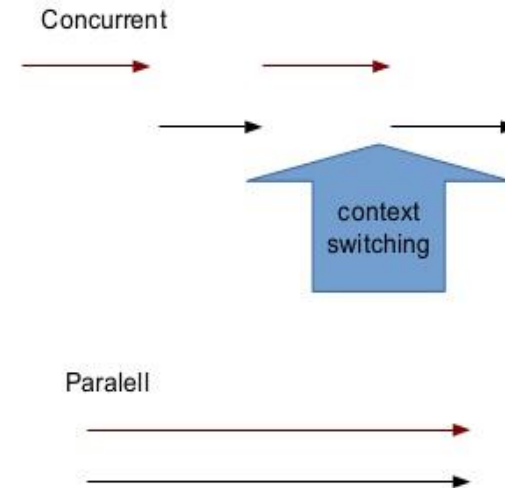
Parallelism vs Concurrency

Parallel programming

- Use additional resources to speed up computation
- Performance perspective

Concurrent programming

- Correct and efficient control of access to shared resources
- Correctness perspective

Distinction is not absolute

Swarnendu Biswas

Approaches to Parallelism

- Multithreading – "assembly language of parallel programming"
- New inherently-parallel languages (e.g., Cilk Plus, X10, and Chapel)
  - New concepts, difficult to get widespread acceptance
- Language extensions (e.g., OpenMP)
  - Easy to extend, but requires compiler or preprocessor support
- Library (e.g., C++ STL and Intel TBB)
  - Works with existing environments, usually no new compiler is needed

Swarnendu Biswas

Challenges with a multithreaded implementation

- Oversubscription or undersubscription, scheduling policy, load imbalance, portability
  - For example, mapping of logical to physical threads is crucial
  - Mapping also depends on whether computation waits on external devices
- Non-trivial impact of time slicing with context switches, cache cooling effects, and lock preemption
  - Time slicing allows more logical threads than physical threads

Swarnendu Biswas

Task-Based Programming

- Programming at the abstraction of tasks is an appealing alternative
- A task is a sequence of instructions (logical unit of work) that can be processed concurrently with other tasks in the same program
  - Interleaving of tasks is constrained by control and data dependences
  - Tasks are lighter-weight compared to logical threads

Swarnendu Biswas

Intel Threading Building Blocks

Swarnendu Biswas

What is Intel TBB?

- A **library** to help leverage multicore performance using standard C++
  - Does not require programmers to be an expert
    - Writing a correct and scalable parallel loop is not straightforward
  - Does not require support for new languages and compilers
  - Does not directly support vectorization

- TBB was first available in 2006
  - Current release is 2020 Update 3
  - Open source and licensed versions available

Swarnendu Biswas

What is Intel TBB?

- TBB works at the abstraction of **tasks** instead of low-level threads
  - **Specify tasks** that can run concurrently instead of threads
  - Specify work (i.e., tasks), instead of focusing on workers (i.e., threads)
    - Raw threads are like assembly language of parallel programming
  - Maps tasks onto physical threads, efficiently using cache and balancing load
  - Full support for nested parallelism

Swarnendu Biswas

Advantages with Intel TBB

- Promotes scalable data-parallel programming
  - Data parallelism is more scalable than functional parallelism
  - Functional blocks are usually limited while data parallelism scales with more processors
  - Not tailored for I/O-bound or real-time processing
- Compatible with other threading packages and is portable
  - Can be used in concert with native threads and OpenMP
  - Relies on generic programming (e.g., C++ STL)

Key Features of Intel TBB

Generic Parallel algorithms
```
parallel_for, parallel_for_each,
parallel_reduce, parallel_scan,
parallel_do, pipeline,
parallel_pipeline, parallel_sort,
parallel_invoke
```

Task scheduler
```
task_group, structured_task_group,
task, task_scheduler_init
```

Concurrent containers
```
concurrent_hash_map
concurrent_unordered_map
concurrent_queue
concurrent_bounded_queue
concurrent_vector
```

Synchronization primitives
atomic operations, condition_variable
various flavors of mutexes

Utilities
```
tick_count
tbb_thread
```

Memory allocators
```
tbb_allocator, cache_aligned_allocator, scalable_allocator,
zero_allocator
```

Swarnendu Biswas

Task-Based Programming with Intel TBB

- Intel TBB parallel algorithms map tasks onto threads automatically
  - Task scheduler manages the thread pool
- Oversubscription and undersubscription of core resources is prevented by task-stealing technique of TBB scheduler

Swarnendu Biswas

An Example: Parallel loop
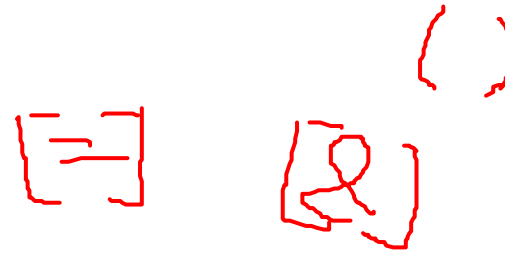
```cpp
#include <chrono>
#include <iostream>
#include <tbb/parallel_for.h>
#include <tbb/tbb.h>

using namespace std;
using namespace std::chrono;
using HRTimer = high_resolution_clock::time_point;

#define N (1 << 26)

void seq_incr(float* a) {
  for (int i = 0; i < N; i++) {
    a[i] += 10;
  }
}
```

```cpp
void parallel_incr(float* a) {
  tbb::parallel_for(static_cast<size_t>(0),
 static_cast<size_t>(N),
  [&](size_t i) {
    a[i] += 10;
  });
}
```

Swarnendu Biswas

An Example: Parallel loop

```cpp
int main() {
  float* a = new float[N];
  for (int i = 0; i < N; i++) {
    a[i] = static_cast<float>(i);
  }

  HRTimer start = high_resolution_clock:
:now();
  seq_incr(a);
  HRTimer end = high_resolution_clock::n
ow();
  auto duration = duration_cast<microsec
onds>(end - start).count();
  cout << "Sequential increment in " <<
duration << " us\n";

  start = high_resolution_clock::now();
  parallel_incr(a);
  end = high_resolution_clock::now();
  duration = duration_cast<microseconds>
(end - start).count();
  cout << "Intel TBB Parallel increment
in " << duration << " us\n";

  return EXIT_SUCCESS;
}
```
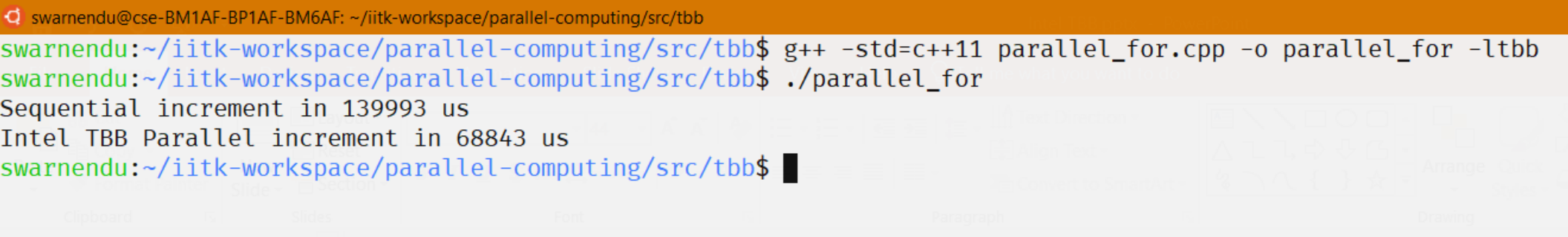
An Example: Parallel loop

```
int main() {
  float* a = new float[N];
  for (int i = 0; i < N; i++) {
```

```
start = high_resolution_clock::now();
parallel_incr(a);
end = high_resolution_clock::now();
```

```
;
  seq_incr(a);
  HRTimer end = high_resolution_clock::n
ow();
  auto duration = duration_cast<microsec
onds>(end - start).count();
  cout << "Sequential increment in " <<
duration << " us\n";
```

```
  return EXIT_SUCCESS;
}
```

Initializing the TBB Library

```
#include <tbb/task_scheduler_init.h>

using namespace tbb;


int main( ) {
    task_scheduler_init init;
    ...
    return 0;
}
```

Not required in recent versions, >= TBB 2.2

- Control when the task scheduler is constructed and destroyed
- Specify the number of threads used by the task scheduler
- Specify the stack size for worker threads

Swarnendu Biswas

Pthreads vs Intel TBB

Pthreads	Intel TBB
• Low-level wrapper over OS support for threads	• Provides high-level constructs and parallel patterns

OpenMP vs Intel TBB

OpenMP	Intel TBB
• Language extension consisting of pragmas, routines, and environment variables	• Library for task-based programming
• Supports C, C++, and Fortran	• Supports C++ with generics
• User can control scheduling policies	• Automated divide-and-conquer approach to scheduling, with work stealing
• OpenMP limited to specified types (for e.g., reduction)	• Generic programming is flexible with types

Swarnendu Biswas

Generic Parallel Algorithms

Swarnendu Biswas

Generic Programming

- Enables distribution of useful high-quality algorithms and data structures

- Write best possible algorithm with fewest constraints (for e.g., `std::sort`)

- Instantiate algorithm to specific situation
  - C++ template instantiation, partial specialization, and inlining make resulting code efficient

Swarnendu Biswas

Generic Programming Example

- The compiler creates the needed versions

T must define a copy constructor and a destructor

```
template <typename T> T max (T x, T y) {
  if (x < y) return y;
  return x;
}

int main() {
  int i = max(20,5);
  double f = max(2.5, 5.2);
  MyClass m = max(MyClass("foo"), MyClass("bar"));
  return 0;
}
```

T must define operator <

Swarnendu Biswas

Intel Threading Building Blocks Patterns

- High-level parallel and scalable patterns

parallel_for	load-balanced parallel execution of independent loop iterations
parallel_reduce	load-balanced parallel execution of independent loop iterations that perform reduction
parallel_scan	template function that computes prefix scan in parallel (y[i] = y[i-1] op x[i])
parallel_while	load-balanced parallel execution of independent loop iterations with unknown or dynamically changing bounds
pipeline	data-flow pipeline pattern
parallel_sort	parallel sort

Swarnendu Biswas

parallel_for

```
void SerialApplyFoo(float a[], size_t n) {
  for (size_t i=0; i<n; ++i)
    foo(a[i]);
}
```

Swarnendu Biswas

Class Definition for `parallel_for`

```
#include "tbb/blocked_range.h"
#include ...

class ApplyFoo {
    float *const m_a;
public:
    void operator()(const blocked_range<size_t>& r) const {
        float *a = m_a;
        for (size_t i=r.begin(); i!=r.end( ); ++i)
            foo(a[i]);
    }
    ApplyFoo(float a[]) : m_a(a) {}
};
```

Body object

Task

Swarnendu Biswas

parallel_for

```
#include "tbb/parallel_for.h"

void ParallelApplyFoo(float a[], size_t n) {
  parallel_for(blocked_range<size_t>(0,n,grainSize), ApplyFoo(a));
}
```

- `parallel_for` schedules tasks to operate in parallel on subranges of the original iteration space using available threads
  - Work is load balanced across the available processors
  - Available cache is used efficiently (similar to tiling)
  - Adding more processors improves performance of existing code

Swarnendu Biswas

Requirements for `parallel_for` Body

- The object has to have a copy constructor and destructor if memory is dynamically allocated
  - `Body::Body(const Body&)`
  - `Body::~Body()`
- `operator()` should not modify the body
  - `parallel_for` requires that the body object's `operator()` be declared as `const`
  - Apply the body to a subrange
    - `void Body::operator() (Range& subrange) const`

Swarnendu Biswas

Example 1

```cpp
class ParallelAverage {
  const float* m_input;
  float* m_output;

public:
  ParallelAverage(float* a, float* b) : m_input(a), m_output(b) {}

  void operator()(const blocked_range<int>& range) const {
    for (int i = range.begin(); i != range.end(); ++i)
      m_output[i] = (m_input[i - 1] + m_input[i] + m_input[i + 1]) * (1 / 3.0f);
  }
};
…
ParallelAverage avg(a, par_out);
parallel_for(blocked_range<int>(1, N - 1), avg);
```

Swarnendu Biswas

Example 1'

```
parallel_for(static_cast<int>(1), static_cast<int>(N – 1),
    [&](int i) {
        lamda_out[i] = (a[i – 1] + a[i] + a[i + 1]) * (1 / 3.0f);
    });
```
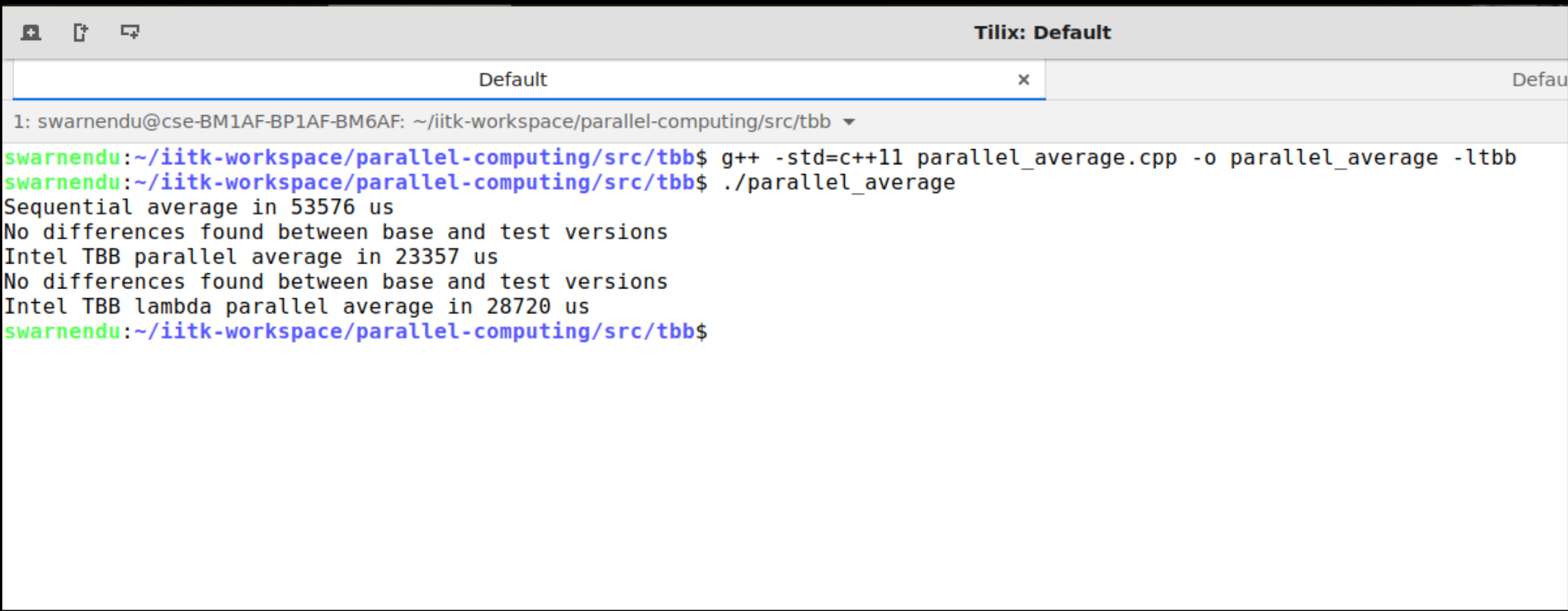

```
// Compile:
g++ –std=c++11 parallel_average.cpp -o parallel_average –ltbb
```

Swarnendu Biswas

Example 1'

```
parallel_for(static_cast<int>(1), static_cast<int>(N - 1),
    [&](int i) {
```

Default   × | Defau

1: swarnendu@cse-BM1AF-BP1AF-BM6AF: ~/iitk-workspace/parallel-computing/src/tbb ▾

```
swarnendu:~/iitk-workspace/parallel-computing/src/tbb$ g++ -std=c++11 parallel_average.cpp -o parallel_average -ltbb
swarnendu:~/iitk-workspace/parallel-computing/src/tbb$ ./parallel_average
Sequential average in 53576 us
No differences found between base and test versions
Intel TBB parallel average in 23357 us
No differences found between base and test versions
Intel TBB lambda parallel average in 28720 us
swarnendu:~/iitk-workspace/parallel-computing/src/tbb$
```

Co

g-

Swarnendu Biswas

Splittable Concept

- A type is splittable if it has a splitting constructor that allows an instance to be split into two pieces
- `X::X(X& x, tbb::split)`
  - Split x into x and a newly constructed object
  - Attempt to split x roughly into two non-empty halves
  - Set x to be the first half, and the constructed object is the second half
  - Dummy argument distinguishes from a copy constructor
- Used in two contexts
  - Partition a range into two subranges that can be processed concurrently
  - Fork a body (function object) into two bodies that can run concurrently

Swarnendu Biswas

Range is Generic

- `R::R(const R&)`

- `R::~R()`

- `bool R::is_divisible() const`

- `bool R::empty() const`

- `R::R(R& r, split)`

- Copy constructor

- Destructor

- True if splitting constructor can be called, false otherwise

- True if range is empty, false otherwise

- Splitting constructor. It splits range r into two subranges. One of the subranges is the newly constructed range. The other subrange is overwritten onto r.

Swarnendu Biswas

More about Ranges

- `tbb::blocked_range<int>(0,8)` represents the index range {0,1,2,3,4,5,6,7}

```
// Construct half-open interval [0,30) with grainsize of 20
blocked_range<int> r(0,30,20);
assert(r.is_divisible());
// Call splitting constructor
blocked_range<int> s(r);
// Now r=[0,15) and s=[15,30) and both have a grainsize 20, inherited from the original value of r
assert(!r.is_divisible());
assert(!s.is_divisible());
```
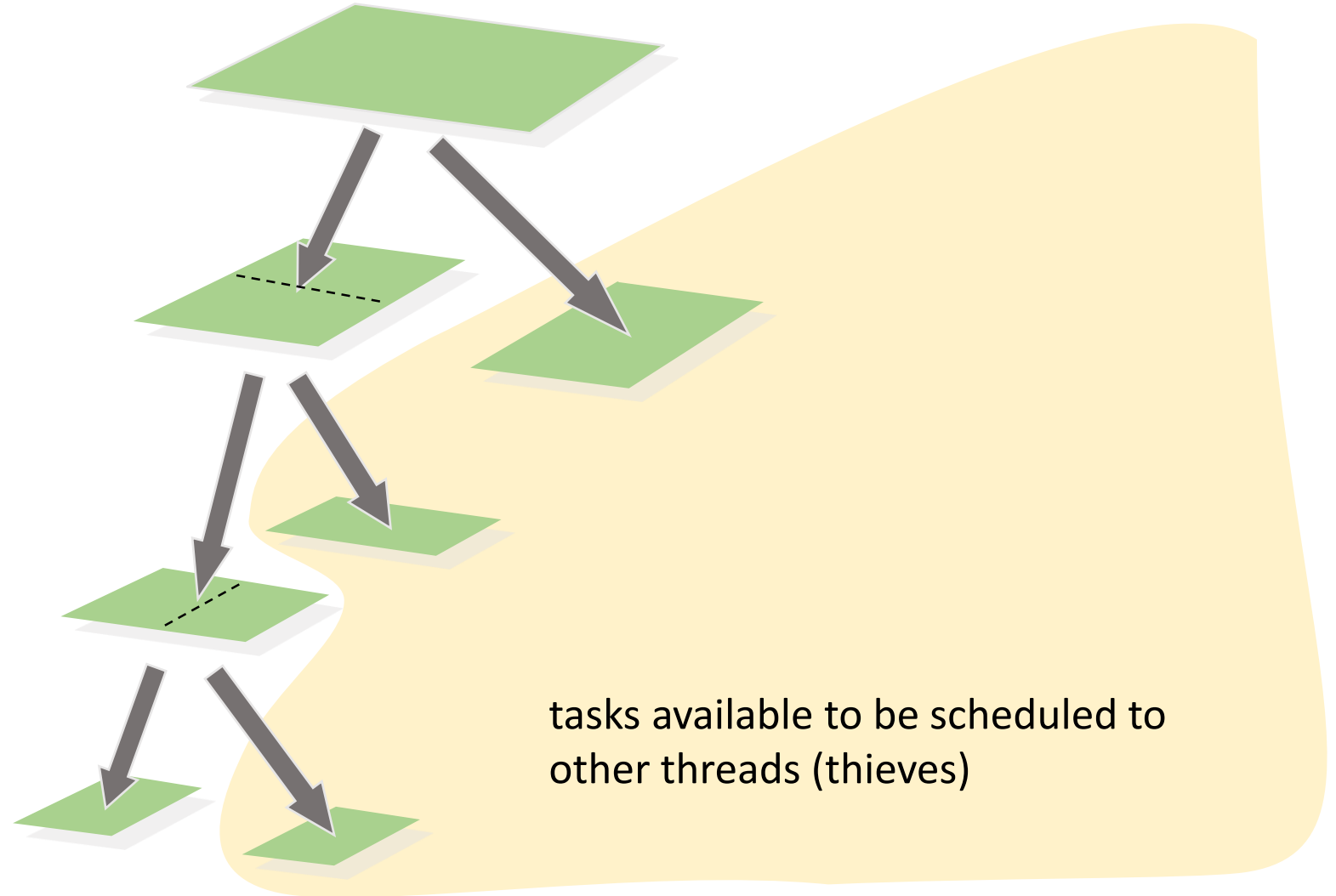
Swarnendu Biswas

More about Ranges

- A two-dimensional variant is `tbb::blocked_range2d`

- Permits using a single `parallel_for` to iterate over two dimensions at once

- Can yield better cache behavior than nesting two one-dimensional instances of `parallel_for`

Splitting over 2D Range

tasks available to be scheduled to other threads (thieves)

Swarnendu Biswas

Grain Size

- Specifies the number of iterations for a chunk to give to a processor
- Impacts parallel scheduling overhead

Swarnendu Biswas

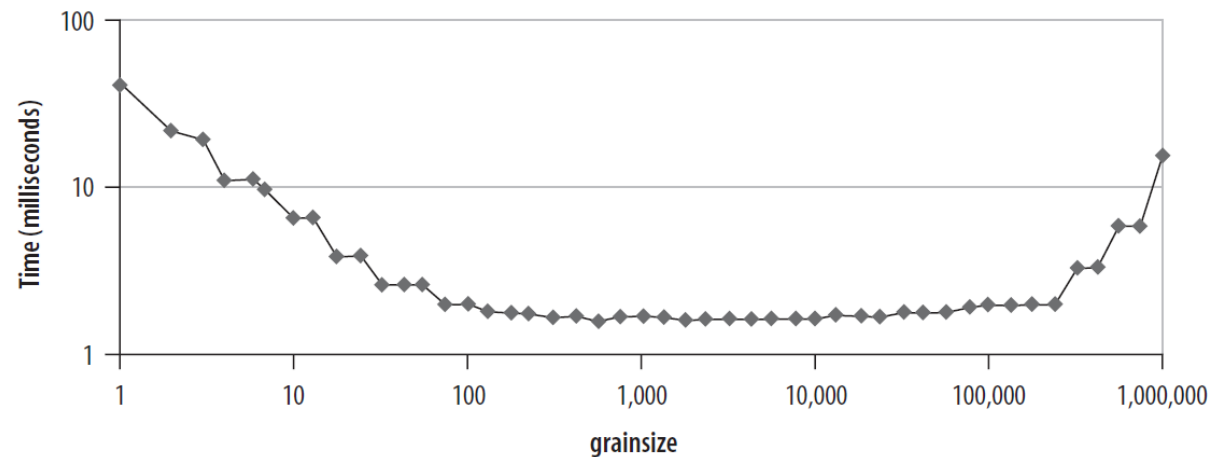# Set the Right Grain Size

- Set the grainsize parameter higher than necessary

- Run your algorithm on one processor core

- Start halving the grainsize parameter

- See how much the algorithm slows down as the value decreases

Partitioner

- Range form of `parallel_for` takes an optional partitioner argument

  `parallel_for(range,bodyobject,simple_partitioner());`

- `auto_partitioner`: Runtime will try to subdivide the range to balance load, this is the default
- `simple_partitioner`: Runtime will subdivide the range into subranges as finely as possible; method `is_divisible` will be false for the final subranges
- `affinity_partitioner`: Request that the assignment of subranges to underlying threads be similar to a previous invocation of `parallel_for` or `parallel_reduce` with the same `affinity_partitioner` object

Swarnendu Biswas

Affinity Partitioner

- When can the affinity partitioner be useful?
  - The computation does a few operations per data access
  - The data acted upon by the loop fits in cache
  - The loop, or a similar loop, is re-executed over the same data

```
void ParallelApplyFoo(float a[], size_t n) {
  static affinity_partitioner ap; // Lives across loop iterations
  parallel_for(blocked_range<size_t>(0,n), ApplyFoo(a), ap);
}
void TimeStepFoo(float a[], size_t n, int steps) {
  for (int t=0; t<steps; ++t)
    ParallelApplyFoo(a, n);
}
```

Swarnendu Biswas

Partitioners

Partitioner	Description	Iteration Space
simple_partitioner	Chunk size bounded by grain size	$\lceil g/2 \rceil \leq chunksize \leq g$
auto_partitioner (default)	Automatic chunk size	$\lceil g/2 \rceil \leq chunksize$
affinity_partitioner	Automatic chunk size and cache affinity	

Swarnendu Biswas

parallel_reduce

- #include <tbb/parallel_reduce.h>

  Value tbb::parallel_reduce(range, identity, func,
  reduction [, partitioner…])

  - Apply func to subranges in range and reduce the results using the binary operator reduction
  - Parameters func and reduction can be lambda expressions

- void parallel_reduce(range, body, [, partitioner…]

Serial Reduction

```
float SerialSumFoo(float a[], size_t n) {
  float sum = 0;
  for (size_t i=0; i!=n; ++i)
    sum += Foo(a[i]);
  return sum;
}
```

Swarnendu Biswas

Parallel Reduction

Assume iterations are independent

```
float ParallelSumFoo(const float *a, size_t n) {
  SumFoo sf(a);
  parallel_reduce(blocked_range<size_t>(0,n), sf);
  return sf.my_sum;
}
```

Parallel Reduction

```
class SumFoo {
  float* my_a;
public:
  float my_sum;

  void operator()(const
            blocked_range<size_t>& r) {
    float *a = my_a;
    float sum = my_sum;
    for (size_t i=r.begin(); i!=r.end(); ++i)
      sum += Foo(a[i]);
    my_sum = sum;
  }
```

```
  SumFoo(SumFoo& x, split) : my_a(x.my_a),
                             my_sum(0.0f)
  {}

  void join(const SumFoo& y) {
    my_sum += y.my_sum;
  }

  SumFoo(float a[]) : my_a(a),
                      my_sum(0.0f)
  {}
};
```

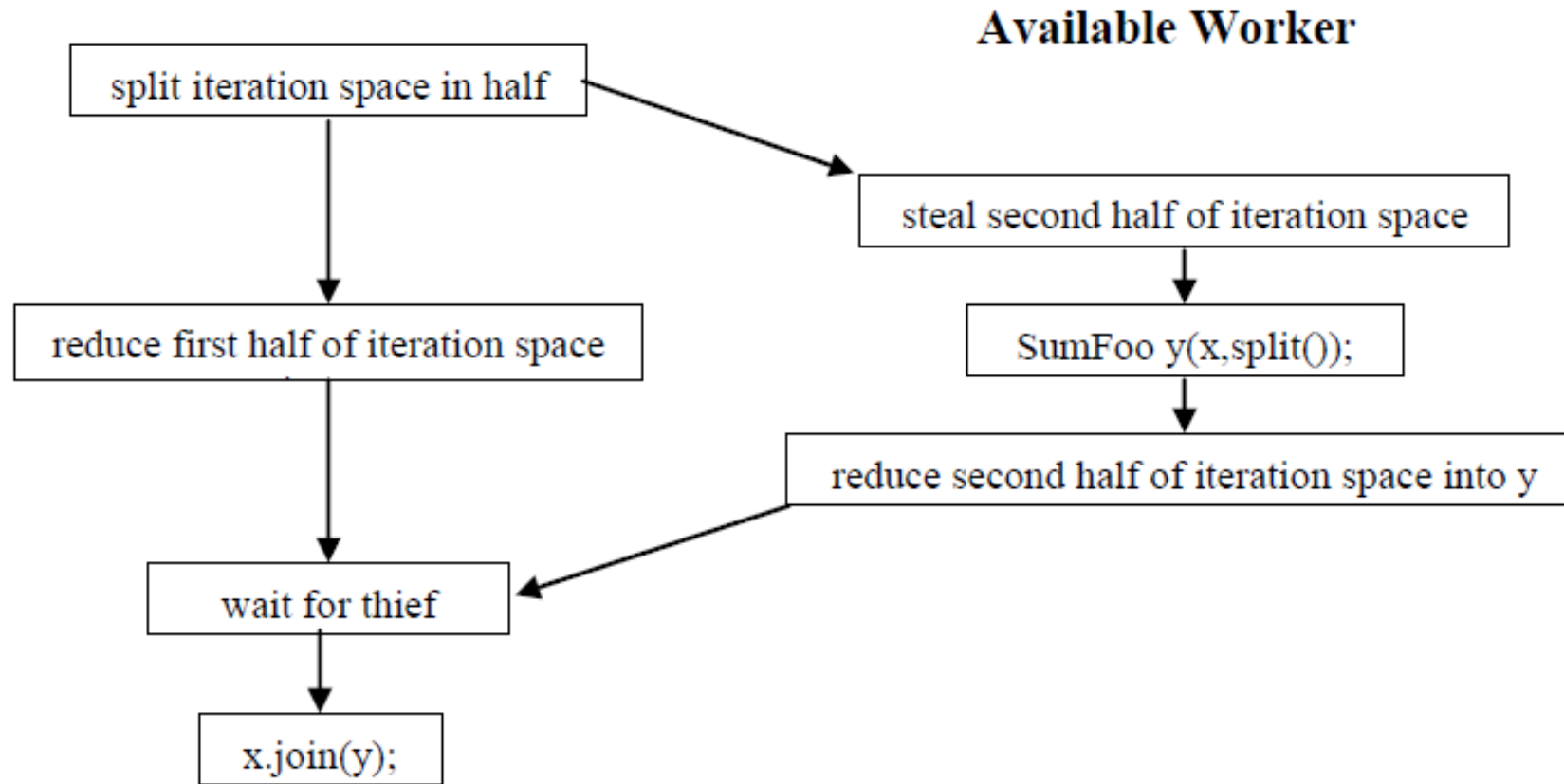# Differences between Parallel For and Reduce

parallel_for	parallel_reduce

- `operator()` is constant

- Requires only a copy ctor

- `operator()` is not constant

- Requires a splitting ctor for creating subtasks

- Requires a `join()` function to accumulate the results of the subtasks

Swarnendu Biswas

Graph of the Split-Join Sequence

Available Worker

split iteration space in half

steal second half of iteration space

reduce first half of iteration space

SumFoo y(x,split());

reduce second half of iteration space into y

wait for thief

x.join(y);

Swarnendu Biswas

One Possible Execution of `parallel_reduce`

`blocked_range<int>(0, 20, 5);`

$$b_0[0,20)$$

$$b_0[0,10) \qquad b_2[10,20)$$

$$b_0[0,5) \qquad b_1[5,10) \qquad b_2[10,15) \qquad b_3[15,20)$$
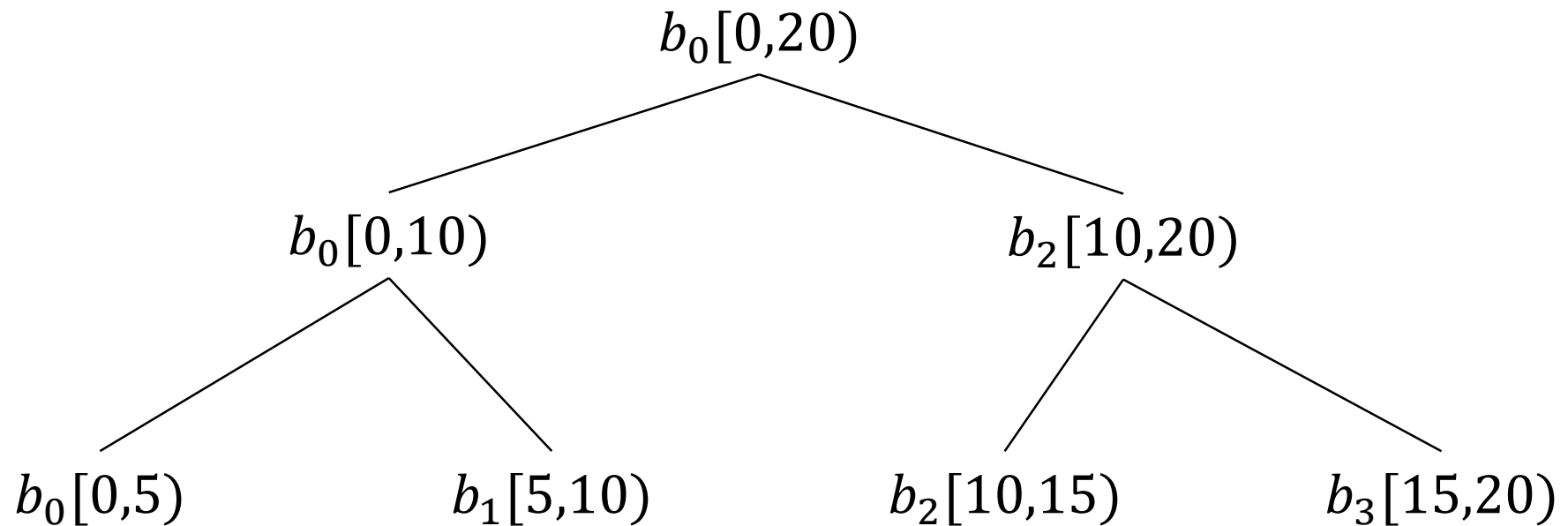
Another Possible Execution of `parallel_reduce`

`blocked_range<int>(0, 20, 5);`

$b_0[0,20)$

$b_0[0,10)$ $b_2[10,20)$

$b_0[0,5)$ $b_1[5,10)$ $b_2[10,15)$ $b_2[15,20)$

Swarnendu Biswas
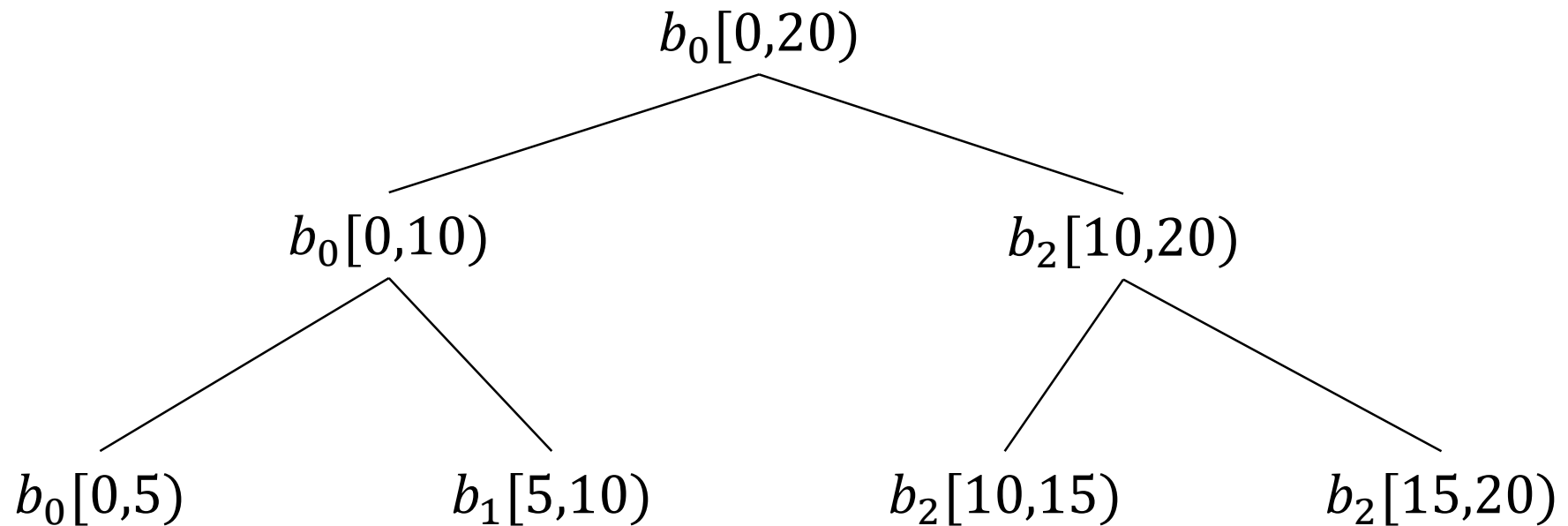
Graph of the Split-Join Sequence
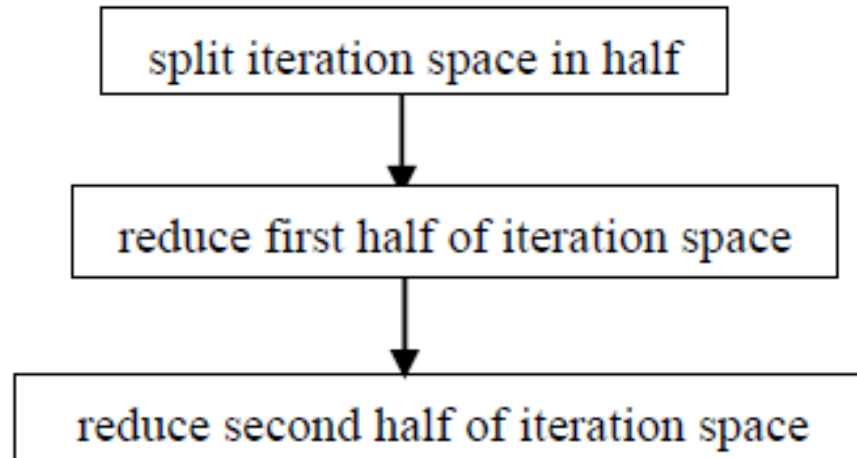
No Available Worker

Swarnendu Biswas

Incorrect Definition of Parallel Reduction

```cpp
class SumFoo {
  float* my_a;
public:
  float my_sum;

  void operator()(const
blocked_range<size_t>& r) {
    float *a = my_a;
    float sum = 0; // WRONG
    size_t end = r.end();
    for (size_t i=r.begin(); i!=end; ++i)
      sum += Foo(a[i]);
    my_sum = sum;
  }
```

```cpp
  SumFoo(SumFoo& x, split) : my_a(x.my_a),
                             my_sum(0) {}

  void join(const SumFoo& y) {
    my_sum+=y.my_sum;
  }

  SumFoo(float a[]) : my_a(a), my_sum(0) {}
};
```

Swarnendu Biswas

Tasks and Task Scheduler

Swarnendu Biswas

TBB Task Scheduler

- Parallel algorithms make use of the task scheduler
  - TBB parallel algorithms map tasks onto threads automatically
  - Task scheduler manages the thread pool
    - Scheduler is *unfair* to favor tasks that have been most recent in the cache

| Problem	TBB Approach
Oversubscription	One scheduler thread per hardware thread
Fair scheduling	Non-preemptive unfair scheduling
High overhead	Programmer specifies tasks, not threads
Load imbalance	Work stealing balances load

Swarnendu Biswas

Task-Based Programming

Serial Code

```
long SerialFib(long n) {
  if (n < 2)
    return n;
  else
    return SerialFib(n-1) +
SerialFib(n-2);
}
```

Task Graph for Fibonacci Calculation

Swarnendu Biswas
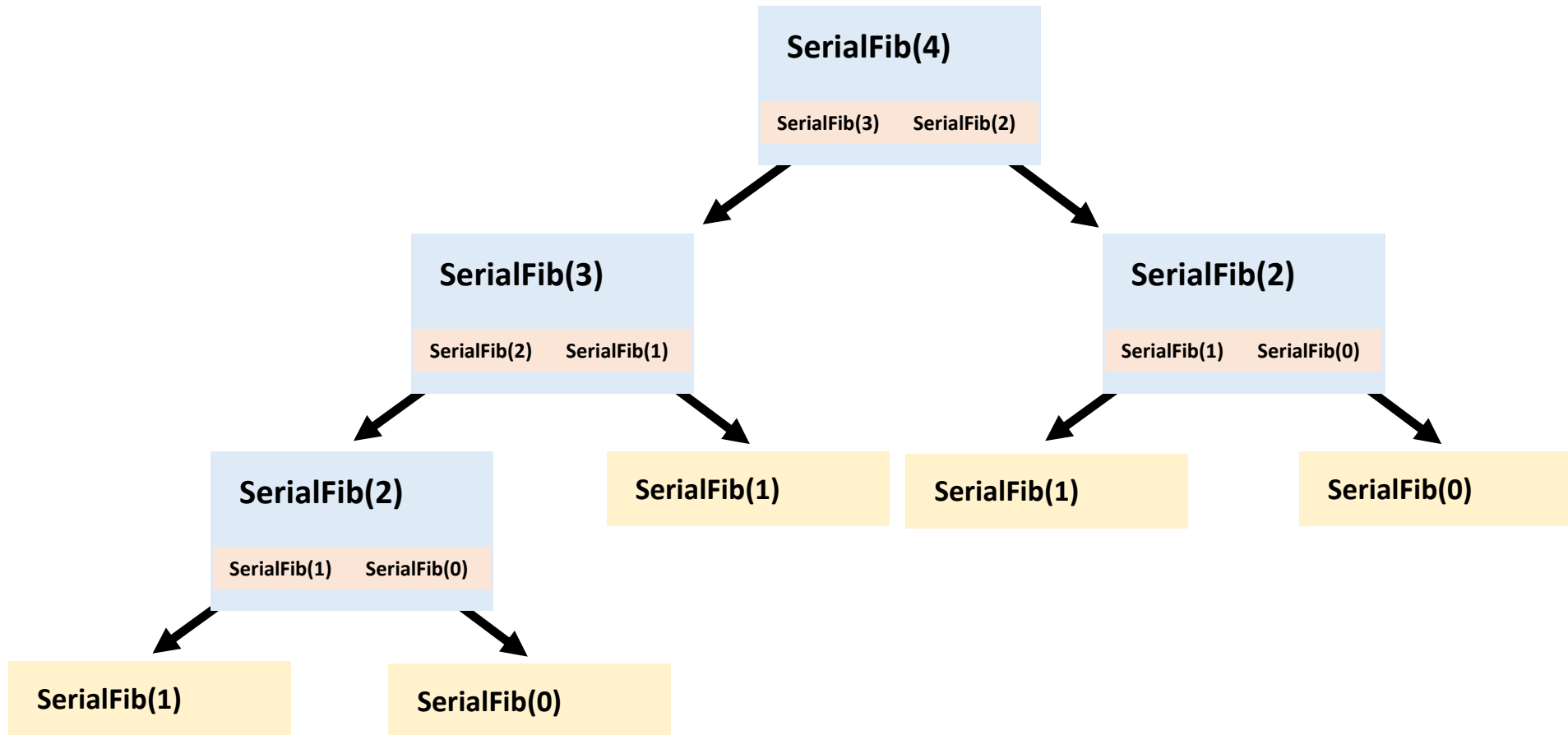
Task-Based Fibonacci

Serial Code	TBB Code

```
long SerialFib(long n) {
  if (n < 2)
    return n;
  else
    return SerialFib(n-1) +
SerialFib(n-2);
}
```

```
long ParallelFib(long n) {
  long sum;
  FibTask& a =
*new(task::allocate_root())
FibTask(n,&sum);
  task::spawn_root_and_wait(a);
  return sum;
}
```

Description of FibTask Class

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

  task* execute() {
    if (n<CutOff) {
      *sum = SerialFib(n);
    }
```

```
    else {
      long x, y;
      FibTask& a = *new(allocate_child())
FibTask(n-1,&x);
      FibTask& b = *new(allocate_child())
FibTask(n-2,&y);
      // 2 children + 1 for the wait
      set_ref_count(3);
      spawn(b); // Return immediately
      spawn_and_wait_for_all(a);
      *sum = x+y;
    }
    return NULL;
}};
```

Task Scheduler

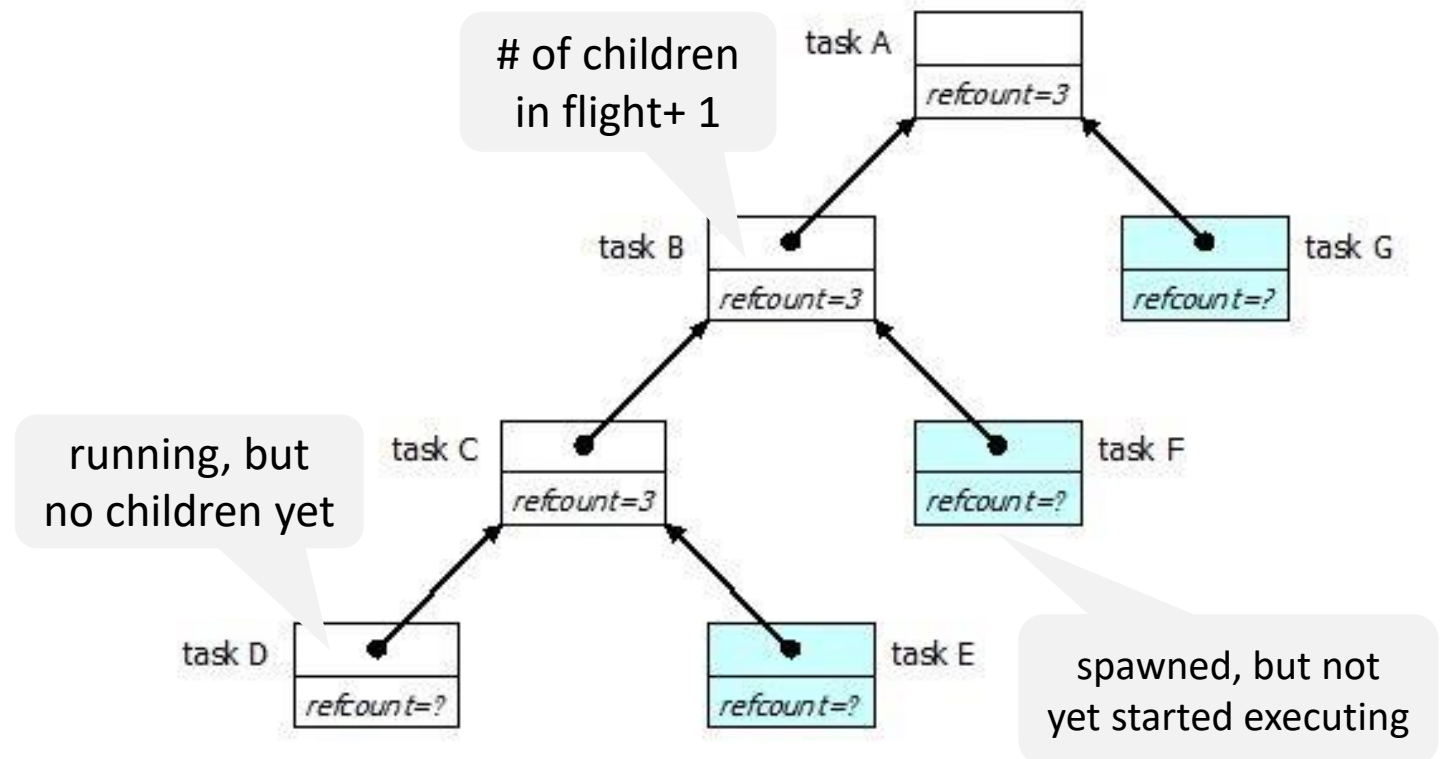- Engine that drives the parallel algorithms and task groups

- Each task has a method `execute()`
  - Definition should do the work of the task
  - Return either `NULL` or a pointer to the next task to run

- Once a thread starts running `execute()`, the task is bound to that thread until `execute()` returns
  - During that period, the thread serves other tasks only when it has to wait for some event

How Task Scheduling Works

- Scheduler evaluates a task graph

- Each task has a refcount
  - Number of tasks that have it as a successor

# of children in flight+ 1

running, but no children yet

spawned, but not yet started executing

task A
refcount=3

task G
refcount=?

task B
refcount=3

task C
refcount=3

task F
refcount=?

task D
refcount=?

task E
refcount=?

Task Scheduling

- Depth-first execution
  - Deeper tasks are more recently created, and will probably have better locality
  - Sequential execution of the task graph is more memory efficient
- Breadth-first execution
  - Can have more parallelism if more physical threads are available
- TBB scheduler implements a hybrid of depth-first and breadth-first execution

Swarnendu Biswas

Scheduling Algorithm

- Each thread has a "ready pool" of tasks it can run
  - The pool is basically a deque of task objects
- When a thread spawns a task, it pushes it to the end of its own deque

top (oldest task)

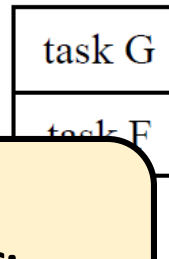| task G |
| --- |
| task F |
| task E |

bottom (youngest task)
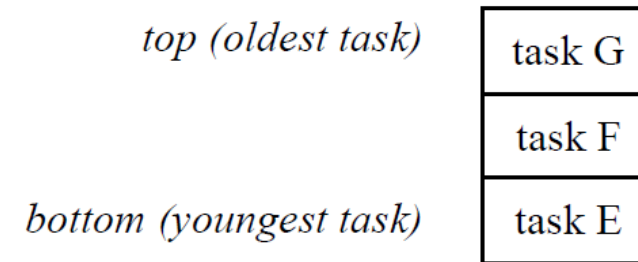
- A thread participates in task graph evaluation
  - Get the task returned by `execute()` for the previous task if any
  - Pops a task from the bottom of its deque
  - Steals a task from the top of another randomly deque

Swarnendu Biswas

Scheduling Algorithm

- There is a shared queue of tasks that were created

- Each thread has a "ready pool" of tasks i
  - The obj

top (oldest task)   task G

task F

Work done is depth-first and stealing is breadth-first

- When a thread spawns a task, it pushes it to the end of its own deque

- Thread participates in task graph evaluation
  - Pops a task from the bottom of its deque
  - Steals a task from the top of another randomly deque

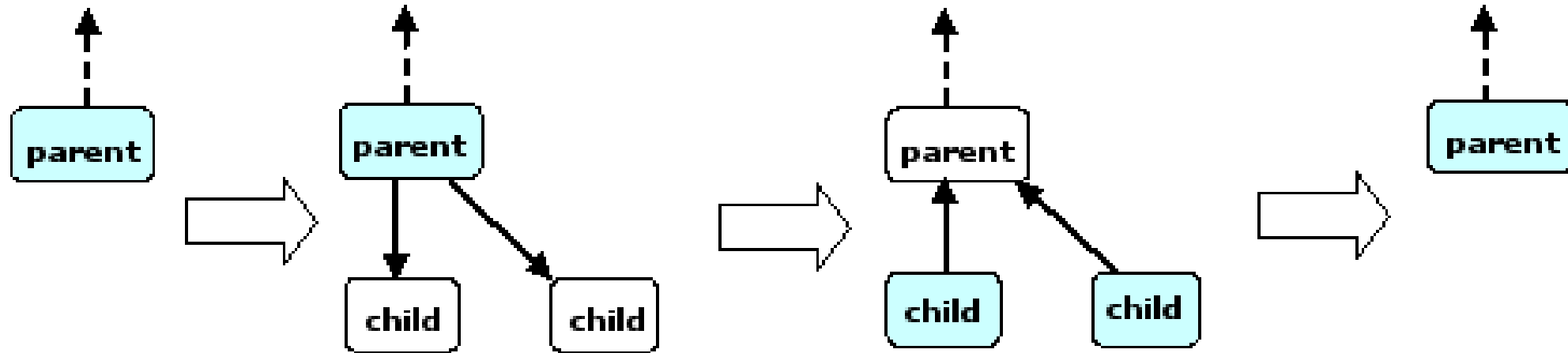# Parallelism in TBB

- Parallelism is generated by split/join pattern
  - Continuation-passing style and blocking style

Swarnendu Biswas

Blocking Style

https://www.threadingbuildingblocks.org/docs/help/reference/task_scheduler.html

Swarnendu Biswas

Disadvantages with Blocking Style

- Worker thread that encounters `wait_for_all()` in parent task is doing no work

- The local variables of a blocked parent task live on the stack
  - Task is not destroyed until all its child are done, problematic for large workloads

Continuation Passing Style

- Concept used in functional programming
- Parent task creates child tasks and specifies a continuation task to be executed when the children complete
  - Continuation inherits the parent's ancestor
- The parent task then exits; it does not block on its children
- The children subsequently run
- After the children (or their continuations) finish, the continuation task starts running
  - Any idle thread can run the continuation task

Swarnendu Biswas

Continuation Passing Style

FibTask with Continuation Passing Style

```
struct FibC: public task {
  long* const sum;
  long x, y;
  F
  FibC(long* sum_) {
    sum = sum_;
  }
  task* execute() {
    *sum = x+y;
    return NULL;
  }
}
```

```
struct FibTask: public task {
  task* execute() {
    if (n < cutOff) { …
    } else {
      FibC& c = *new(allocate_continuation)
FibC(sum);
      FibTask& a = *new(c.allocate_child())
FibTask(n-1,&c.x);
      FibTask& b = *new(c.allocate_child())
FibTask(n-2,&c.y);
      c.set_ref_count(2);
      spawn(b); // Return immediately
      spawn(a);
    }
    return NULL;
}};
```

Scheduler Bypass

```
struct FibTask: public task {
  task* execute() {
    if (n < cutOff) { …
    } else {
      FibC& c = *new(allocate_continuation)
FibC(sum);
      FibTask& a = *new(c.allocate_child())
FibTask(n-1,&c.x);
      FibTask& b = *new(c.allocate_child())
FibTask(n-2,&c.y);
      c.set_ref_count(2);
      spawn(b); // Return immediately
      spawn(a);
    }
    return NULL;
  }};
```

```
struct FibTask: public task {
  task* execute() {
    if (n < cutOff) { …
    } else {
      FibC& c = *new(allocate_continuation)
FibC(sum);
      FibTask& a = *new(c.allocate_child())
FibTask(n-1,&c.x);
      FibTask& b = *new(c.allocate_child())
FibTask(n-2,&c.y);
      c.set_ref_count(2);
      spawn(b); // Return immediately
      return &a;
    }
}};
```

Swarnendu Biswas

Did Tasks Help?

```
class FibTask: public task {                        else {

public:                                                 long x, y;
   const                                                              ;
   long*                                                              ;
   FibTas                                                             ;
n(n_), s                                                         r the


   task*
   if (
      *s
   }
```

```
[ 76%] Built target tbb_parallel_incr
[ 80%] Built target tbb_parallel_change
[ 83%] Built target transformations_example2
[ 86%] Built target transformations_example1
[ 90%] Built target vectorization-sse
[ 93%] Built target vectorization-avx512
[100%] Built target vectorization1
[100%] Built target vectorization-avx
~/i/p/build (master|+2) $ ./bin/tbb_fibonacci
Sequential Fibonacci in 27970585 us
Task-based Fibonacci in 7554692 us
~/i/p/build (master|+2) $
```

Concurrent Containers

Swarnendu Biswas

Concurrent Containers

- TBB Library provides highly concurrent containers
  - STL containers are not concurrency-friendly: attempt to modify them concurrently can corrupt container
  - Standard practice is to wrap a lock around STL containers
    - Turns container into serial bottleneck
- Library provides fine-grained locking or lockless implementations
  - Can be used with the library, OpenMP, or native threads
  - Worse single-thread performance, but better scalability

Concurrent TBB Containers

- TBB containers offer a high level of concurrency
  - Fine-grained locking
    - Multiple threads operate by locking only portions they really need to lock
    - As long as different threads access different portions, they can proceed concurrently
  - Lock-free techniques
    - Different threads account and correct for the effects of other interfering threads

Swarnendu Biswas

Concurrency-Friendly Interfaces

- Some STL interfaces are inherently not concurrency-friendly
- For example, suppose two threads each execute the following

```
extern std::queue q;
if(!q.empty()) {
    item=q.front();
    q.pop();
}
```

At this instant, another thread might pop last element.

- Solution: `concurrent_queue` has `try_pop()`

Serial vs Concurrent Queue

std::queue	tbb::concurrent_queue

```
extern std::queue<T> serialQ;
T item;
if (!serialQ.empty()) {
  item = serialQ.front();
  serialQ.pop_front();
  // process item
}
```

```
extern concurrent_queue<T> myQ;
T item;
if (myQ.try_pop(item)) {
  // process item
}
```

Swarnendu Biswas

Concurrent Queue Container

- `concurrent_queue<T>`
  - FIFO data structure that permits multiple threads to concurrently push and pop items
  - Method `push(const T&)` places copy of item on back of queue. The method waits until it can succeed without exceeding the queue's capacity.
  - `try_push(item)` pushes `item` only if it would not exceed the queue's capacity
  - `pop(item)` waits until it can succeed
  - Method `try_pop(T&)` pops value if available, otherwise it does nothing
  - If a thread pushes values A and B in order, another thread will see values A and B in order

Swarnendu Biswas

Concurrent Queue Container

- `concurrent_queue<T>`
  - Method `size()` returns signed integer
    - Number of push operations started minus the number of pop operations started
    - If `size()` returns –n, it means n pops await corresponding pushes on an empty queue
  - Method `empty()` returns `size() == 0`
    - May return true if queue is empty, but there are pending pop()

Swarnendu Biswas

Concurrent Queue Container Example

```
#include "tbb/concurrent_queue.h"
using namespace tbb;
int main () {
    concurrent_queue<int> queue;
    int j;
    for (int i = 0; i < 10; i++)
        queue.push(i);
    while (!queue.empty()) {
        queue.pop(&j);
        printf("from queue: %d\n", j);
    }
    return 0;
}
```

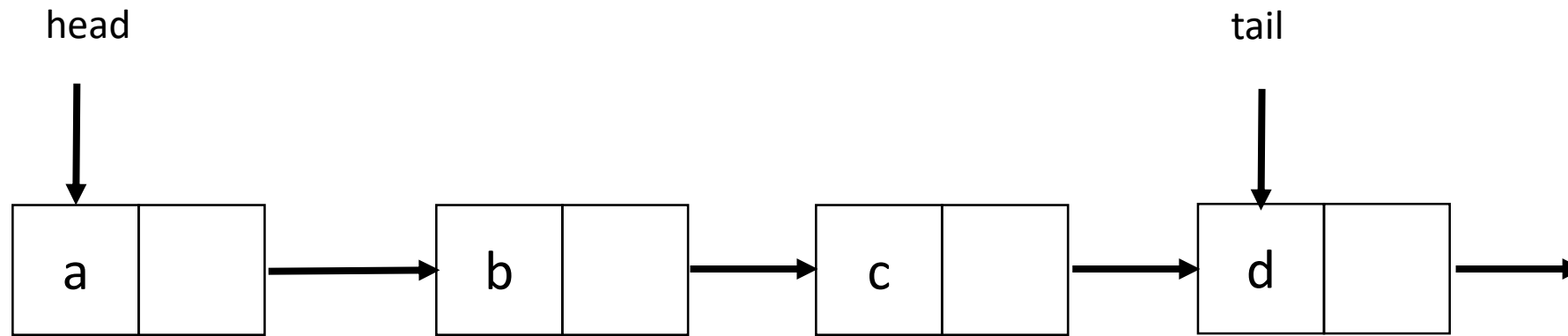- Simple example to enqueue and print integers

ABA Problem

- A thread checks a location to be sure the value is *A* and proceeds with an update only if the value was *A*

- Thread T1 reads value A from shared memory location
- Other threads update A to B, and then back to A
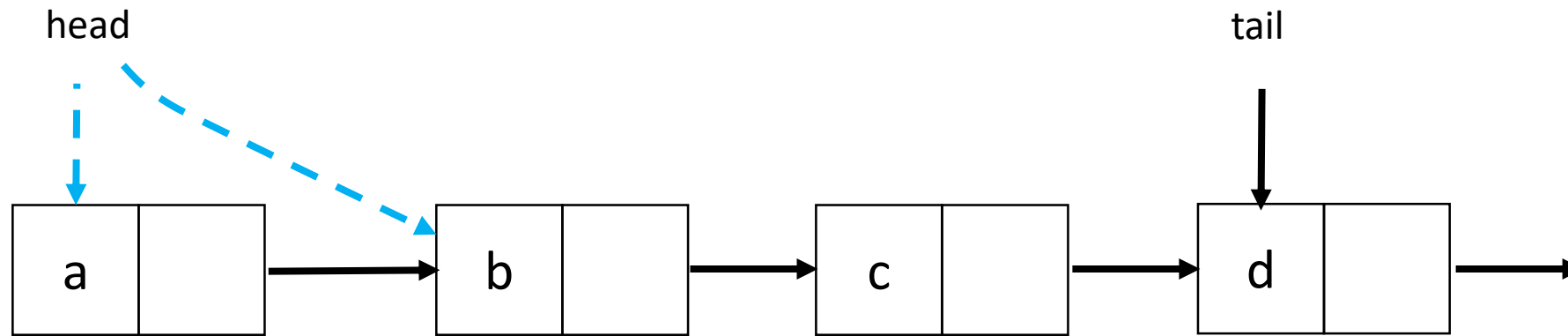- T1 performs `compare_and_swap()` and succeeds

Swarnendu Biswas

Example of ABA Problem

head

tail

a → b → c → d →

- Thread 1 will execute deq(a)

Swarnendu Biswas

Example of ABA Problem

- Thread 1 is executing deq(a), gets delayed

Swarnendu Biswas

Example of ABA Problem

head   tail
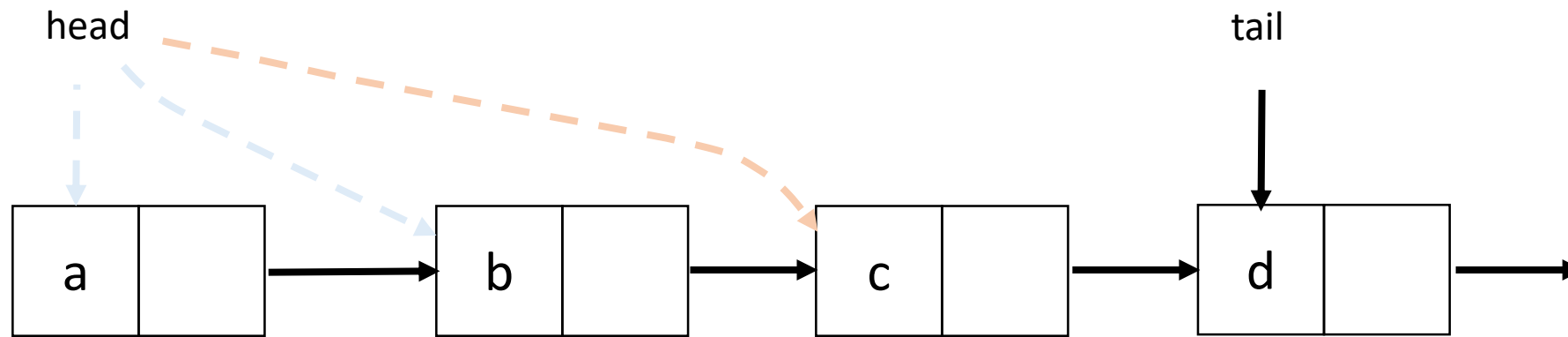
| a | | b | | c | | d | |

- Other threads execute deq(a, b, c, d), then execute enq(a)

Example of ABA Problem

head   tail

a   b

- Other threads execute deq(a, b, c, d), then execute enq(a)

Swarnendu Biswas

Example of ABA Problem
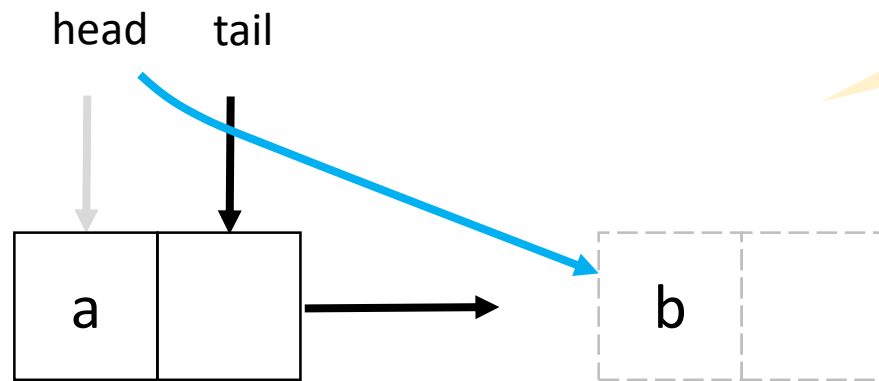
head.compareAndSet(first, next)

head · tail

a → b

- Thread 1 is executes CAS for deq(a), CAS succeeds

Swarnendu Biswas

Concurrent Vector Container

- `concurrent_vector<T>`
  - Dynamically growable array of T
    - Method `grow_by(size_type delta)` appends `delta` elements to end of vector
    - Method `grow_to_at_least(size_type n)` adds elements until vector has at least n elements
    - Method `push_back(x)` safely appends x to the array
    - Method `size()` returns the number of elements in the vector
    - Method `empty()` returns `size() == 0`
  - Never moves elements until cleared
    - Can concurrently access and grow
    - Method `clear()` is not thread-safe with respect to access/resizing

Swarnendu Biswas

Concurrent Vector Container Example

- Append a string to the array of characters held in `concurrent_vector`
  - Grow the vector to accommodate new string
    - `grow_by()` returns old size of vector (first index of new element)
  - Copy string into vector

```
void Append(concurrent_vector<char>& V, const char* string) {
    size_type n = strlen(string)+1;
    memcpy(&V[V.grow_by(n)], string, n+1);
}
```

Swarnendu Biswas

Concurrent HashMap Container

- `concurrent_hash_map<Key,T,HashCompare>`
  - Maps Key to element of type T
  - Define class HashCompare with two methods
    - `hash()` maps Key to hashcode of type size_t
    - `equal()` returns true if two Keys are equal
  - Enables concurrent `find()`, `insert()`, and `erase()` operations
    - An `accessor` grants read-write access
    - A `const_accessor` grants read-only access
    - Lock released when smart pointer is destroyed, or with explicit release()

Swarnendu Biswas

Concurrent HashMap Container Example

```cpp
// Structure that defines hashing and comparison operations for user's type
struct MyHashCompare {
  static size_t hash( const string& x ) {
    size_t h = 0;
    for (const char* s = x.c_str(); *s; ++s )
      h = (h*17)^*s;
    return h;
  }
  static bool equal( const string& x, const string& y ) {
    return x==y;
  }
};
```

Swarnendu Biswas

Concurrent HashMap Container Example

```
// A concurrent hash table that maps strings to ints
typedef concurrent_hash_map<string,int,MyHashCompare> StringTable;
// Function object for counting occurrences of strings
struct Tally {
  StringTable& table;
  Tally(StringTable& table_) : table(table_) {}
  void operator()( const blocked_range<string*> range ) const {
    for (string* p=range.begin(); p!=range.end(); ++p) {
      StringTable::accessor a;
      table.insert(a, *p);
      a->second += 1;
    }
  }
};
```

Swarnendu Biswas

Concurrent HashMap Container Example

```
const size_t N = 1000000;
string Data[N];

void CountOccurrences() {
  StringTable table;
  parallel_for(blocked_range<string*>(Data, Data+N, 1000), Tally(table));

  for (StringTable::iterator i=table.begin(); i!=table.end(); ++i)
    printf("%s %d\n",i->first.c_str(),i->second);
}
```

Swarnendu Biswas

Scalable Memory Allocation

Swarnendu Biswas

Scalable Memory Allocators

- Serial memory allocation can easily become a bottleneck in multithreaded applications
  - Threads require mutual exclusion into shared global heap
  - In the old days, a single-process lock was used for `malloc()` and `free()` in `libc`
  - Many `malloc()` alternatives are now available (`jemalloc()`, `tcmalloc()`)
  - New C++ standards are trying to deal with this
    - Smart pointers, `std::aligned_alloc` (C++17)
- False sharing - threads accessing the same cache line
  - Even accessing distinct locations, cache line can ping-pong

Scalable Memory Allocators

- TBB offers two choices for scalable memory allocation
  - Similar to the STL template class `std::allocator`
  - `scalable_allocator`
    - Offers scalability, but not protection from false sharing
    - Memory is returned to each thread from a separate pool
  - `cache_aligned_allocator`
    - Two objects allocated by this allocator are guaranteed to not have false sharing
    - Always allocates on a cache line, increases space usage

```
std::vector<int, cache_aligned_allocator<int>>
```

Swarnendu Biswas

Methods for `scalable_allocator`

- `#include <tbb/scalable_allocator.h>`

- Scalable versions of malloc, free, realloc, calloc
  - `void *scalable_malloc(size_t size);`
  - `void scalable_free(void *ptr);`
  - `void *scalable_realloc(void *ptr, size_t size);`
  - `void *scalable_calloc(size_t nobj, size_t size);`

Swarnendu Biswas

Synchronization Primitives

Swarnendu Biswas

Synchronization Primitives

- Mutual exclusion is implemented with mutex objects and locks
  - Mutex is the object on which a thread can acquire a lock
- Several mutex variants are available

- Critical regions of code are protected by scoped locks
  - The range of the lock is determined by its lifetime (scope)
  - Does not require the programmer to remember to release the lock
  - Leaving lock scope calls the destructor, making it exception safe

Mutex Example

```
spin_mutex mtx; // Construct unlocked mutex
{
  // Create scoped lock and acquire lock on mtx
  spin_mutex::scoped_lock lk(mtx);
  // Critical section
} // Lock goes out of scope, destructor releases the lock



spin_mutex::scoped_lock lk;
lk.acquire(mtx);
// Critical section
lk.release();
```

Atomic Execution

```
atomic<int> i;
. . .
int z = i.fetch_and_add(2);
```

- `atomic<T>`
  - T should be integral type or pointer type
  - Full type-safe support for 8, 16, 32, and 64-bit integers

Operations	Semantics
"= x" and "x = "	read/write value of x
x.fetch_and_store(y)	z = x, x = y, return z
x.fetch_and_add(y)	z = x, x += y, return z
x.compare_and_swap(y, z)	w = x, if (x == z) { x = y, return w; }

Summary

- Intel Threading Building Blocks is a data parallel programming model for C++ applications
  - Used for computationally intense code
  - Uses generic programming
- Intel Threading Building Blocks provides
  - Generic parallel algorithms
  - Highly concurrent containers
  - Low-level synchronization primitives
  - A task scheduler that can be used directly
- Learn when to use or mix Intel TBB, OpenMP or explicit threading

Swarnendu Biswas

References

- J. Reindeers. Intel Threading Building Blocks Outfitting C++ for Multi-Core Processor Parallelism.

- https://www.threadingbuildingblocks.org/docs/help/index.htm

- Intel. Threading for Performance with Intel Threading Building Blocks

- M. Voss. What's New in Threading Building Blocks. OSCON 2008.

- Vivek Sarkar. Intel Thread Building Blocks. COMP 422, Rice University.

- M. McCool et al. Structured Parallel Programming: Patterns for Efficient Computation.