Parallel Programming Course
Threading Building Blocks (TBB)

Paul Guermonprez
www.Intel-Software-Academic-Program.com
paul.guermonprez@intel.com
Intel Software
2012-03-14

Key Features of TBB

You can specify *tasks* instead of manipulating threads

- TBB maps your logical tasks onto threads with full support for *nested parallelism*

Targets threading for *scalable performance*

- Uses proven efficient parallel patterns
- Uses work-stealing to support the load balance of unknown execution time for tasks.

Open source and licensed versions available on :

- Linux, Windows, and Mac OS X*

Open Source community extended support to :

- FreeBSD*, IA Solaris* and XBox* 360

Limitations

Intel® TBB is not intended for

- I/O bound processing
- Real-time processing

because its task scheduler is unfair, in order to have efficient use of cores.

Components of TBB *(version 4.0)*

Parallel algorithms
parallel_for, parallel_for_each
parallel_reduce
parallel_do
parallel_scan
parallel_pipeline & filters
parallel_sort
parallel_invoke

Flow Graphs
functional nodes
(source, continue, function)
buffering nodes
(buffer, queue, sequencer)
split/join nodes
other nodes

Ranges and partitioners

Tasks & Task groups

Task scheduler

Synchronization primitives
atomic operations
mutexes : classic, recursive, spin, queuing
rw_mutexes : spin, queuing

Thread Local Storage
combinable
enumerable_thread_specific
flattened2d

Concurrent containers
concurrent_hash_map
concurrent_queue
concurrent_bounded_queue
concurrent_priority_queue
concurrent_vector
concurrent_unordered_map
concurrent_unordered_set

Memory allocators
tbb_allocator, cache_aligned_allocator, scalable_allocator

Task-based Programming with Intel® TBB

Tasks are light-weight entities at user-level

- Intel® TBB parallel algorithms maps tasks onto threads automatically
- Task scheduler manages the thread pool
  - Scheduler is *unfair*, to favor tasks that have been most recent in the cache
  - Oversubscription and undersubscription of core resources is prevented by task-stealing technique of scheduler

Intel® TBB Algorithms 1/2

Task scheduler powers high level parallel patterns that are pre-packaged, tested, and tuned for scalability :

- parallel_for, parallel_for_each: load-balanced parallel execution of loop iterations where iterations are independent
- parallel_reduce: load-balanced parallel execution of independent loop iterations that perform reduction (e.g. summation of array elements)
- parallel_scan: load-balanced computation of parallel prefix
- parallel_pipeline: linear pipeline pattern using serial and parallel filters

Intel® TBB Algorithms 2/2

- parallel_do: load-balanced parallel execution of independent loop iterations with ability to add more work during its execution

- parallel_sort: parallel sort

- parallel_invoke: parallel execution of function objects or pointers to functions

The parallel_for algorithm

```
#include <tbb/blocked_range.h>
#include <tbb/parallel_for.h>

template<typename Range, typename Func>
Func parallel_for( const Range& range, const Func& f,
 [, task_group_context& group] )
```

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

template<typename Index, typename Func>
Func parallel_for( Index first, Index_type last
 [, Index step], const Func& f [, task_group_context& group] );
```

parallel_for partitions original range into subranges, and deals out subranges to worker threads in a way that:

- Balances load
- Uses cache efficiently
- Scales

Range is Generic

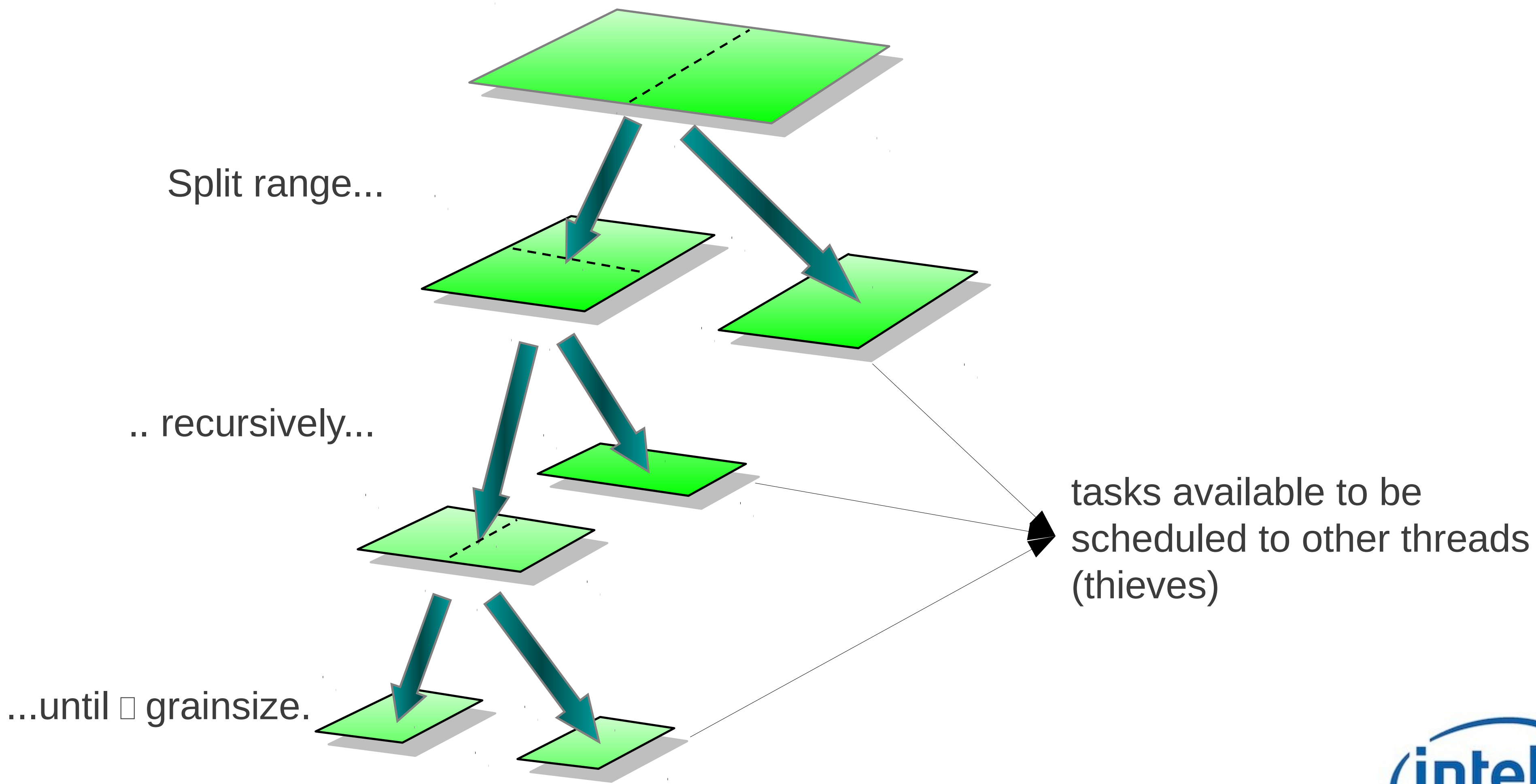
Library provides predefined ranges :

- blocked_range, blocked_range2d, blocked_range3d

You can define yours, it only has to implement these methods :

MyRange::MyRange (const MyRange&)	Copy constructor
MyRange::~MyRange()	Destructor
bool MyRange::is_empty() const	True if range is empty
bool MyRange::is_divisible() const	True if range can be partitioned
MyRange::MyRange(MyRange& r, split)	Splitting constructor; splits r into two subranges

How splitting works on blocked_range2d



Grain Size

OpenMP has similar parameter

Part of blocked_range<>, used by parallel_for and parallel_reduce, not underlying task scheduler

- Grain size exists to amortize overhead, not balance load
- Units of granularity are loop iterations

Typically only need to get it right within an order of magnitude

Tuning Grain Size

- Tune by examining single-processor performance
- When in doubt, err on the side of making it a little too large, so that performance is not hurt when only one core is available.

An Example using parallel_for

Independent iterations and fixed/known bounds

serial code :

```
const int N = 100000;

void change_array(float array, int M) {
    for (int i = 0; i < M; i++){
        array[i] *= 2;
    }
}

int main (){
    float A[N];
    initialize_array(A);
    change_array(A, N);
    return 0;
}
```

An Example using parallel_for

Using the **`parallel_for`** pattern

```
#include <tbb/blocked_range.h>
#include <tbb/parallel_for.h>

using namespace tbb;

void parallel_change_array(float *array, size_t M) {
    parallel_for(blocked_range<size_t>(0, M, IdealGrainSize),
      [=](const blocked_range<size_t>& r) -> void {
        for(size_t i = r.begin(); i != r.end(); i++ )
            array[i] *= 2;
    });
}
```

Task Scheduler

A task scheduler is automatically created when TBB threads are required, and destructed after.

You might want to control that in order to avoid overhead caused by numerous creations/destructions.

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

using namespace tbb;

int main (){
    task_scheduler_init init; //threads creation
    float A[N];
    initialize_array(A);
    parallel_change_array(A, N);
    return 0;
} // out of scope -> threads destruction
```

You can set the maximum number of threads by passing it in argument to the constructor

Generic Programming vs Lambda functions

Generic Programming :

```
class ChangeArrayBody {
    float *array;
public:
    ChangeArrayBody(float *a): array(a) {}
    void operator()( const blocked_range<size_t>& r ) const{
        for (size_t i = r.begin(); i != r.end(); i++ ){
            array[i] *= 2;
        }
    }
};

void parallel_change_array(float *array, size_t M) {
 parallel_for(blocked_range<int>(0, M, IdealGrainSize),
                 ChangeArrayBody(array));
}
```

Lambda functions :

```
void parallel_change_array(float *array, size_t M) {
    parallel_for(blocked_range<size_t>(0, M, IdealGrainSize),
        [=](const blocked_range<size_t>& r) -> void {
            for(size_t i = r.begin(); i != r.end(); i++ )
                array[i] *= 2;
    });
}
```

blue = original code
green = provided by TBB
red = boilerplate for library

Software

Generic Programming vs Lambda functions

You can achieve the same performance with both, but some patterns might require generic programming.

In this course, we will show you the Lambda way when it can be used.

Lambda functions are part of the C++11 standard, you might need to append -std=c++0x to your compiler arguments to use them.

Activity 1: Matrix Multiplication

Convert serial matrix multiplication application into parallel application using parallel_for

- Triple-nested loops

When using Intel® TBB, to automatically configure environment variables such as library path, use :

```
source /opt/intel/bin/compilervars.sh (intel64|ia32)
```

The parallel_reduce algorithm

```
#include <tbb/blocked_range.h>
#include <tbb/parallel_reduce.h>

template<typename Range, typename Value,
 typename Func, typename ReductionFunc>
Value parallel_reduce( const Range& range, const Value& identity,
 const Func& func, const ReductionFunc& reductionFunc,
 [, partitioner[, task_group_context& group]] );
```

parallel_reduce partitions original range into subranges like parallel_for

The function Func is applied on these subranges, the returned result is then merged with the others (or identity if there is none) using the function reductionFunc.

Serial Example

```
#include <limits>

// Find index of smallest element in a[0...n-1]
size_t serialMinIndex( const float a[], size_t n ) {
    float value_of_min = numeric_limits<float>::max();
    size_t index_of_min = 0;
    for( size_t i=0; i<n; ++i ) {
        float value = a[i];
        if( value<value_of_min ) {
            value_of_min = value;
            index_of_min = i;
        }
    }
    return index_of_min;
}
```

Parallel Version

```
#include <limits>
#include <tbb/blocked_range.h>
#include <tbb/parallel_reduce.h>


size_t parallelMinIndex( const float a[], size_t n ) {
    return parallel_reduce(blocked_range<size_t>(0,n,10000),
     size_t(0),
     [=](blocked_range<size_t> &r, size_t index_of_min) -> size_t {
       float value_of_min = a[index_of_min];
       for(size_t i=r.begin();i!=r.end();++i){
           float value = a[i];
           if( value<value_of_min ) {
               value_of_min = value;
               index_of_min = i;
           }
       }
       return index_of_min;
     },
     [=](size_t i1, size_t i2){
       return (a[i1]<a[i2])? i1:i2;
     }
    );
}
```

accumulate result

join

Activity 2: parallel_reduce Lab

Numerical Integration code to compute Pi

Quicksort

Next slides will show you, step by step, how the parallel_sort algorithm execute.

You will see how threads engage in work-stealing : each time a partitions operation divides the segments of values to be sorted, the partitioning thread will take only one; the other is allowed to be picked up by a free thread.

Quicksort – Step 1

```
tbb::parallel_sort (color, color+64);
```

THREAD 1

32 44 9 26 31 57 3 19 55 29 27 1 20 5 42 62 25 51 49 15 54 6 18 48 10 2 60 41 14 47 24 36 37 52 22 34 35 11 28 8 13 43 53 23 61 38 56 16 59 17 50 7 21 45 4 39 33 40 58 12 30 0 46 63

Thread 1 starts with the initial data

Quicksort – Step 2

THREAD 1 THREAD 3 THREAD 2 THREAD 4

32 44 9 26 31 57 3 19 55 29 27 1 20 5 42 62 25 51 49 15 54 6 18 48 10 2 60 41 14 47 24 36 37 52 22 34 35 11 28 8 13 43 53 23 61 38 56 16 59 17 50 7 21 45 4 39 33 40 58 12 30 0 46 63

11 0 9 26 31 30 3 19 12 29 27 1 20 5 33 4 25 21 7 15 17 6 18 16 10 2 23 13 14 8 24 36 32 28 22 34 35

37

52 47 41 43 53 60 61 38 56 48 59 54 50 49 51 45 62 39 42 40 58 55 57 44 46 63

Thread 1 partitions/splits its data

37

Quicksort – Step 2

THREAD 1

THREAD 3

THREAD 2

THREAD 4

32 44 9 26 31 57 3 19 55 29 27 1 20 5 42 62 25 51 49 15 54 6 18 48 10 2 60 41 14 47 24 36 37 52 22 34 35 11 28 8 13 43 53 23 61 38 56 16 59 17 50 7 21 45 4 39 33 40 58 12 30 0 46 63

11 0 9 26 31 30 3 19 12 29 27 1 20 5 33 4 25 21 7 15 17
6 18 16 10 2 23 13 14 8 24 36 32 28 22 34 35

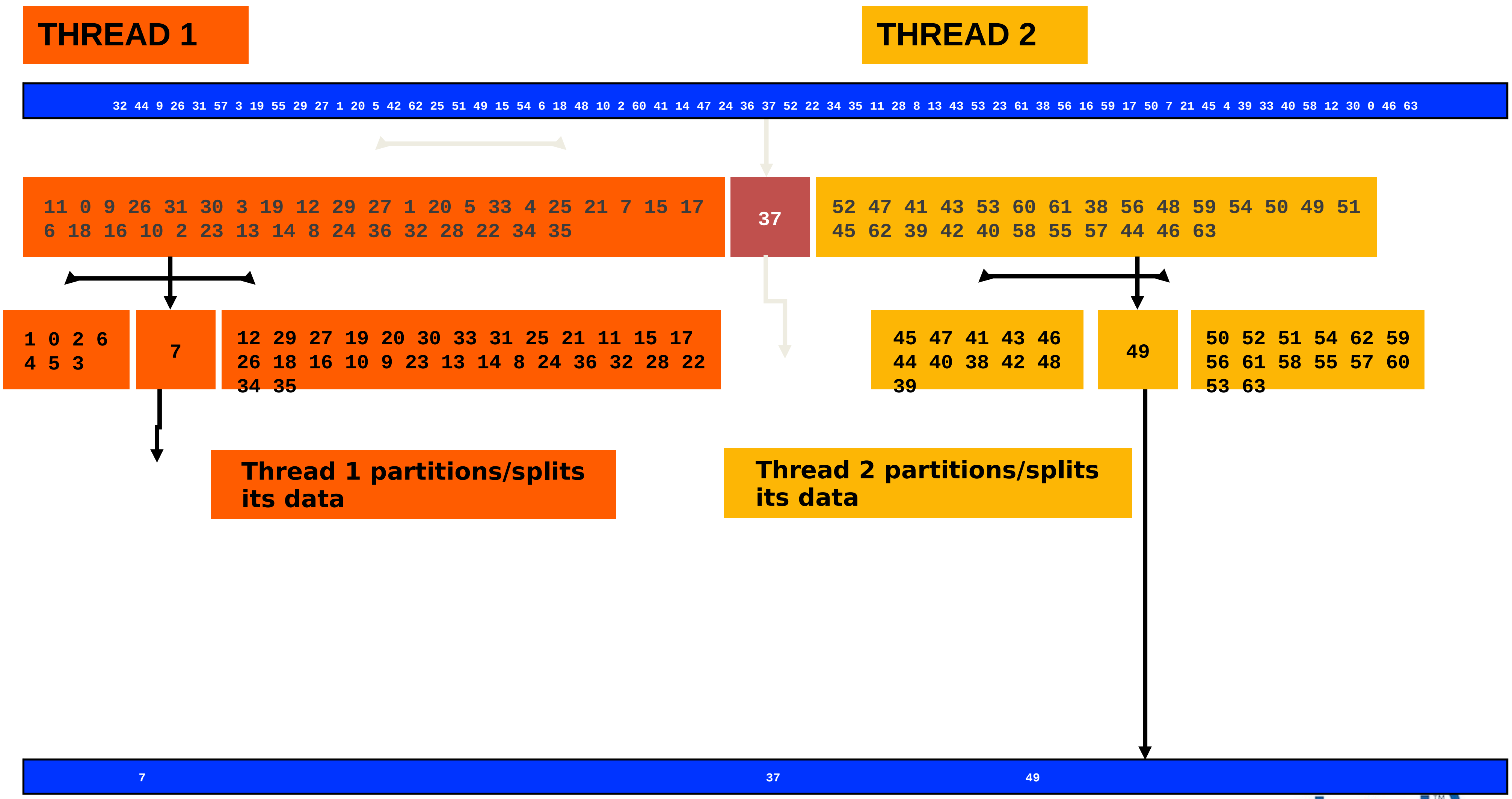
37

52 47 41 43 53 60 61 38 56 48 59 54 50 49 51
45 62 39 42 40 58 55 57 44 46 63

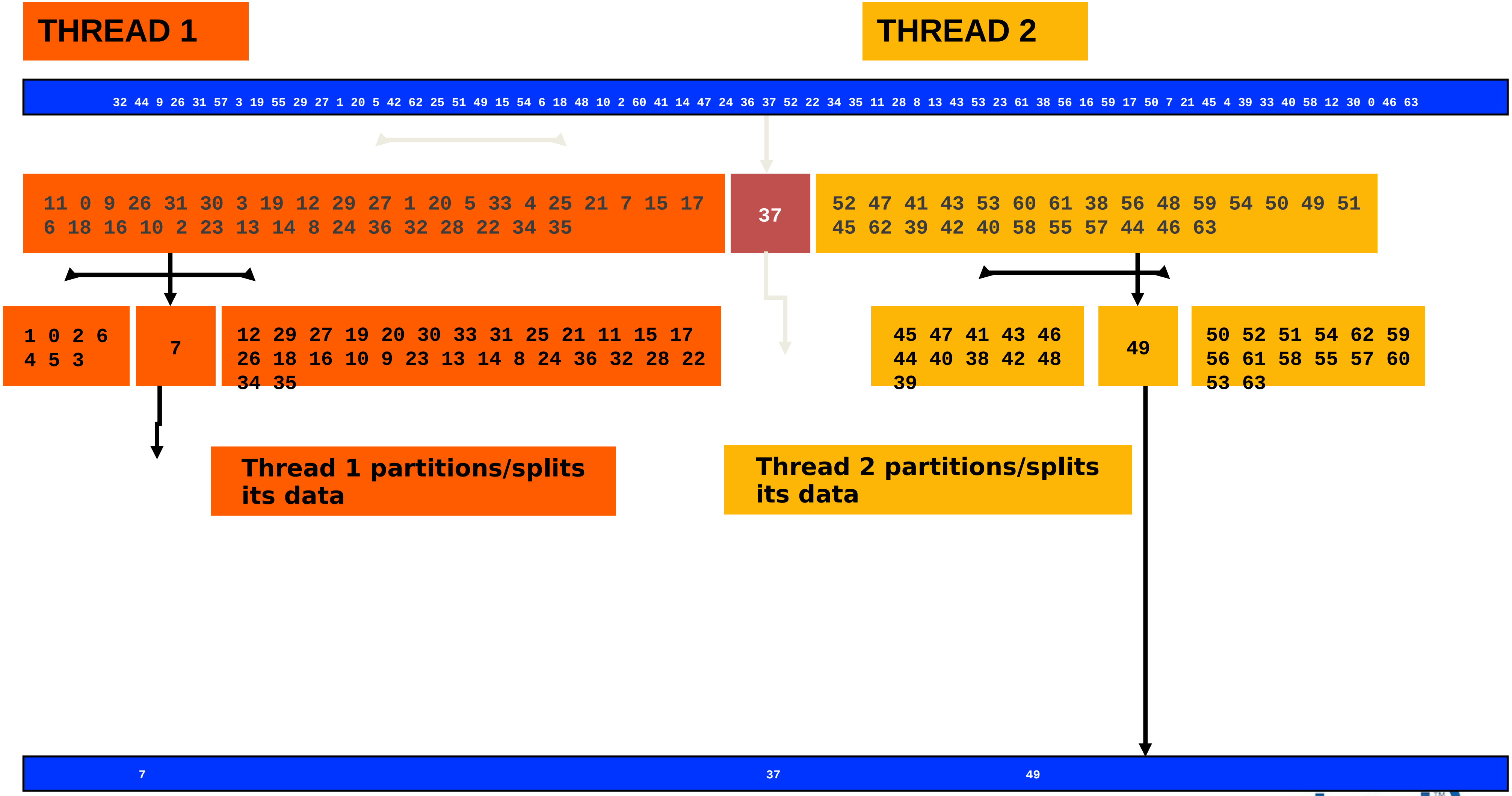
Thread 2 gets work by stealing from Thread 1

37

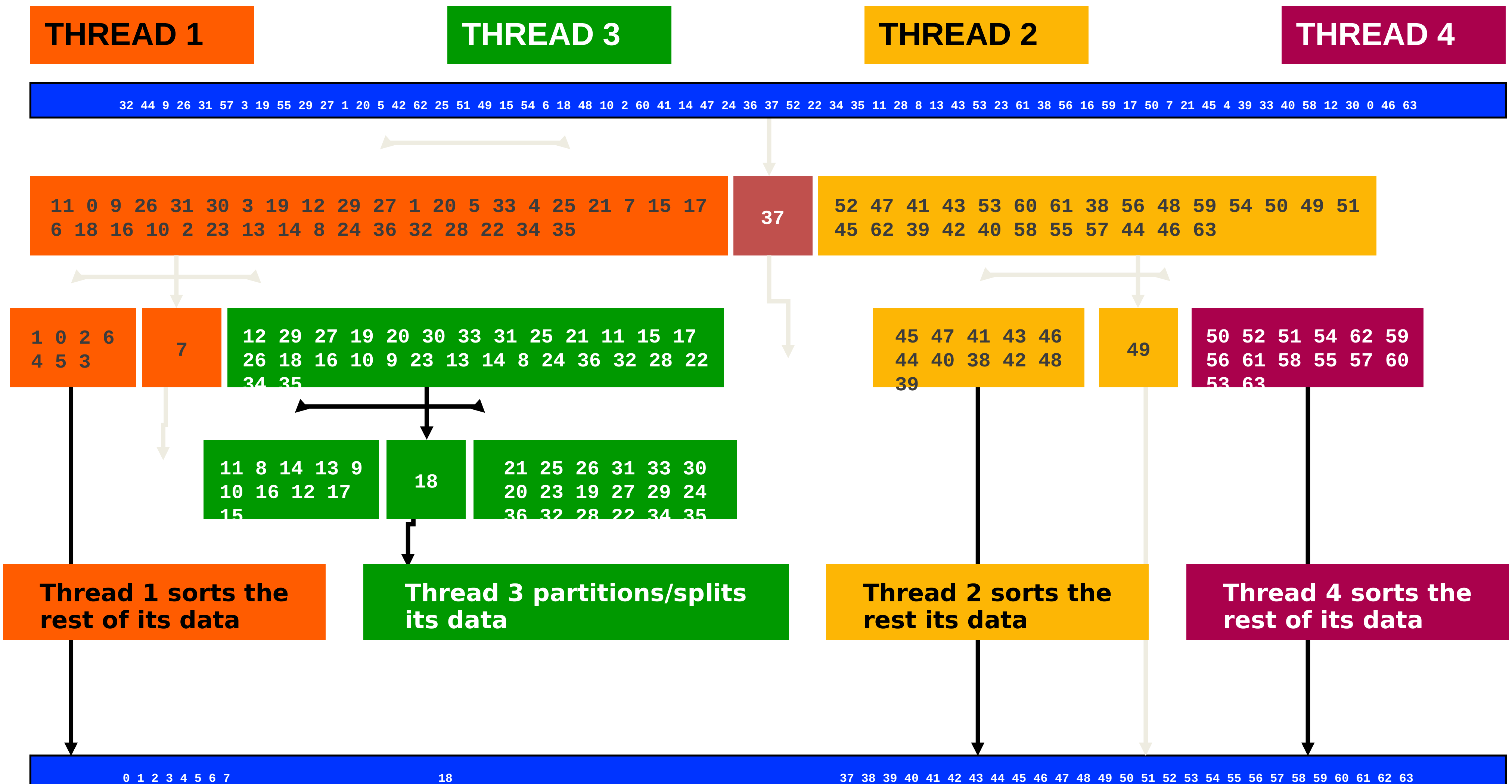
Quicksort – Step 3



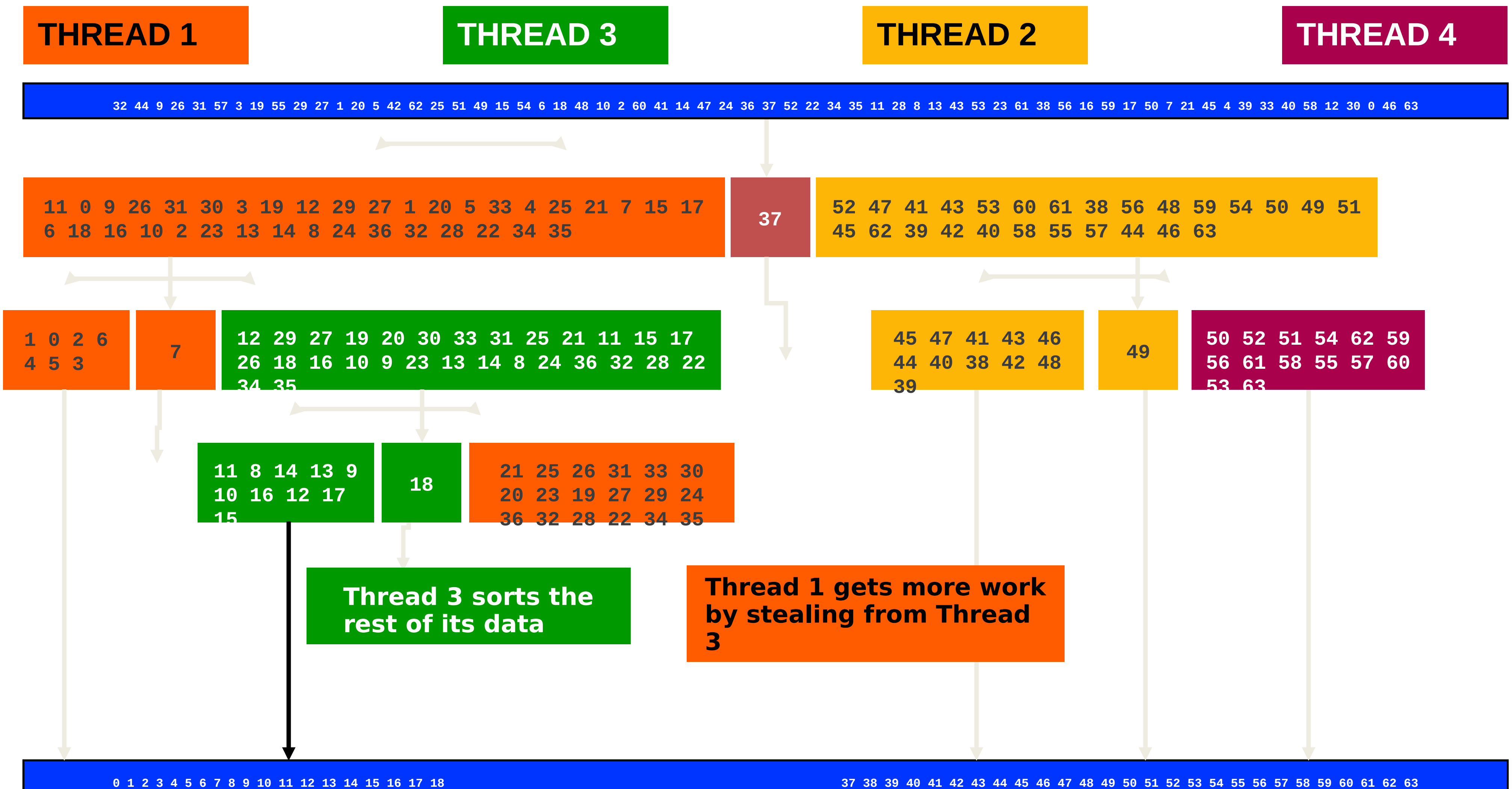
Quicksort – Step 3



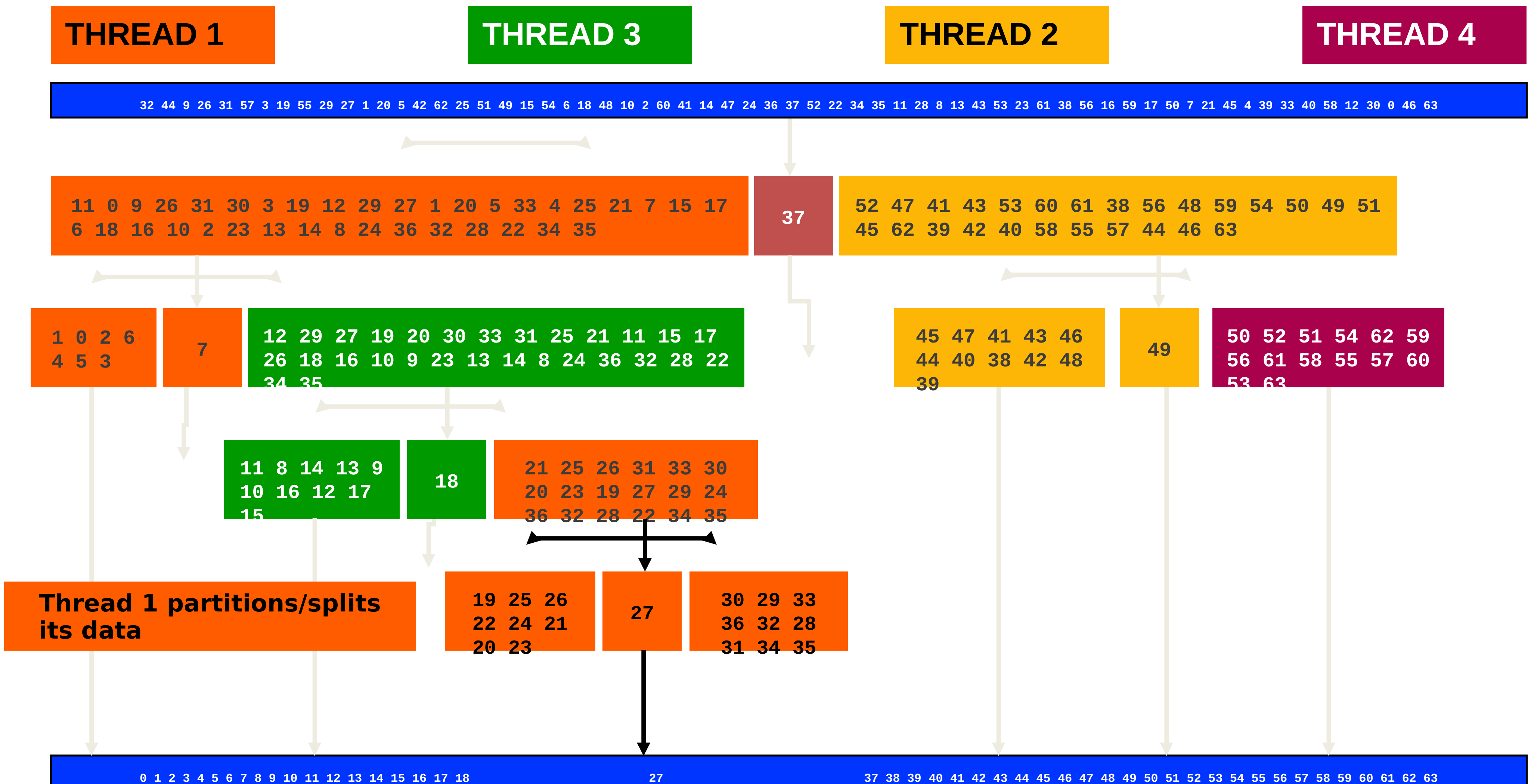
Quicksort – Step 4



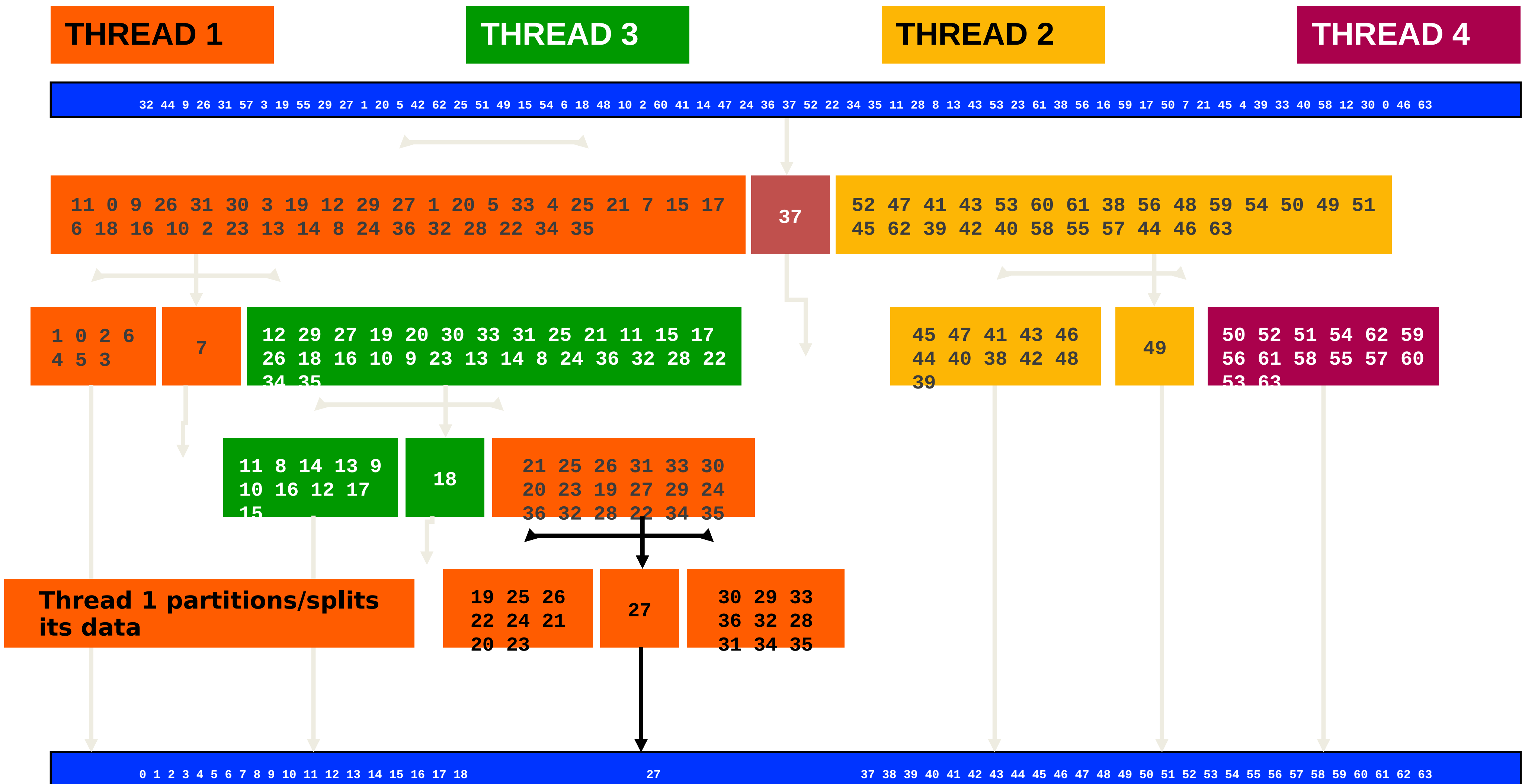
Quicksort – Step 5



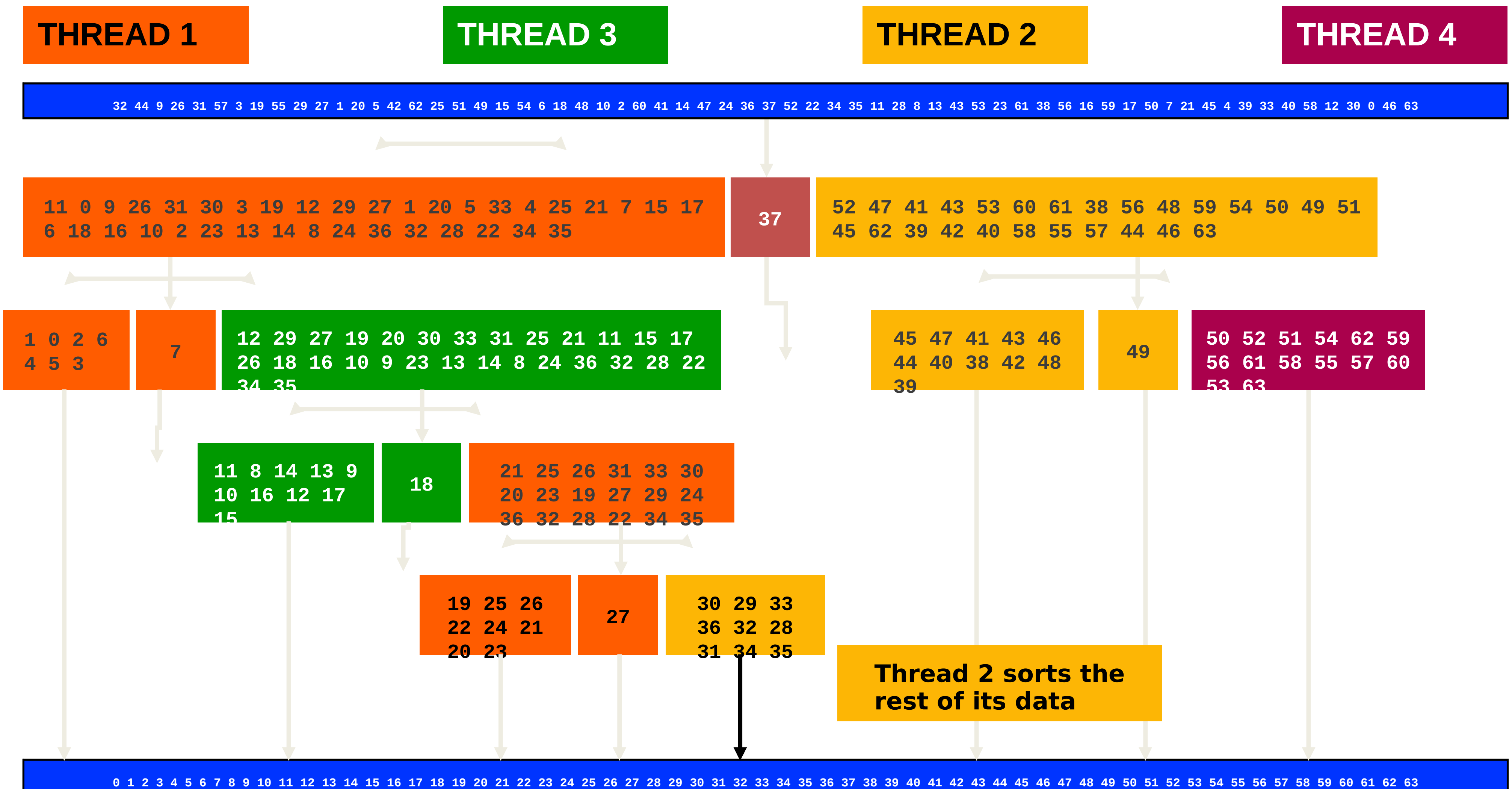
Quicksort – Step 6



Quicksort – Step 6



Quicksort – Step 7



Software

Task Based Approach

Intel® TBB provides C++ constructs that allow you to express parallel solutions in terms of task objects

- Task scheduler manages thread pool
- Task scheduler avoids common performance problems of programming with threads

Problem	Intel® 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

Software

Example: Naive Fibonacci Calculation

Recursion typically used to calculate Fibonacci number

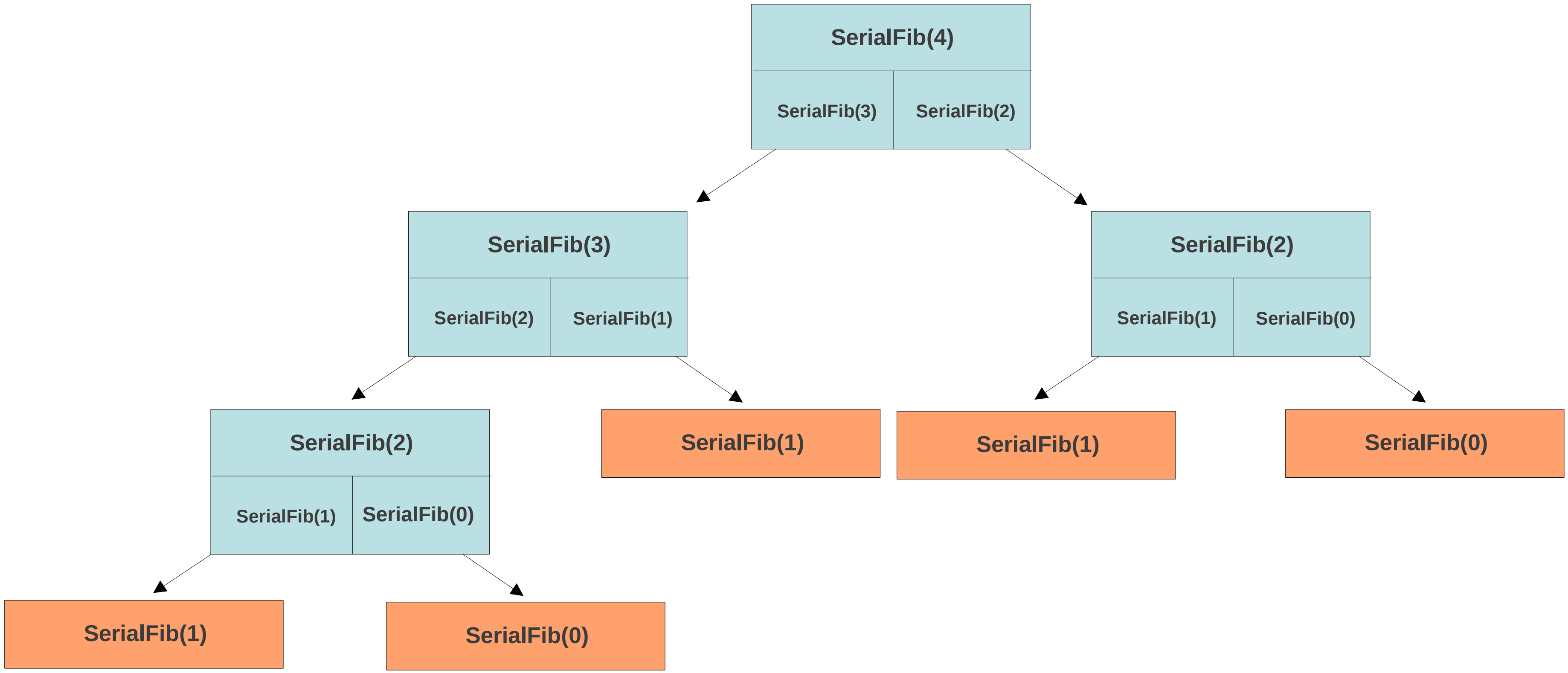
Widely used as toy benchmark

- Easy to code
- Has unbalanced task graph

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

Example: Naive Fibonacci Calculation

Can envision Fibonacci computation as a task graph



Fibonacci - parallel_invoke Solution

```
void parallelFib(int N, long &sum) {
  if (N < 2)
     sum = N;
  else if (N < 1000)
 sum = serialFib(N);
  else {
     long x, y;
     tbb::parallel_invoke(
       [&] () { parallelFib(N-1, x);},
       [&] () { parallelFib(N-2, y);}
     );
     sum = x + y;
  }
}
```

Function you pass cannot have parameters and return value.

It's easily worked around by using lambda capture functionnality – but if you cannot use lambda functions, see the next slides presenting task spawning solution.

Software

Fibonacci - Task Spawning Solution

Use TBB tasks to thread creation and execution of task graph

Create new root task

- Allocate task object
- Construct task

Spawn (execute) task, wait for completion

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

Fibonacci - Task Spawning Solution

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

  task* execute() {     // Overrides virtual function 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);
        set_ref_count(3); // 3 = 2 children + 1 for wait // ref_count
is used to keep track of the number of tasks spawned at the current
level of the task graph
        spawn( b );
        spawn_and_wait_for_all( a ); //set tasks for execution and
wait for them
        *sum = x+y;
    }
    return NULL;
  }
};
```

Activity 3: Task Scheduler Interface for Recursive Algorithms

Develop code to launch Intel TBB tasks to traverse and sum a binary tree.

Implement parallel_invoke and/or task spawning solutions.

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

- Worse single-thread performance, but better scalability.
- Don't need the task scheduler - can be used with the library, OpenMP, or native threads.

Concurrency-Friendly Interfaces

Some STL interfaces are inherently not concurrency-friendly

For example, suppose two threads each execute:

```
extern std::queue q;
if(!q.empty()) {
    //At this instant, another thread might pop
        last element and queue becomes empty
    item=q.front();
    q.pop();
}
```

Solution: concurrent_queue has pop_if_present

Concurrent Queue Container

concurrent_queue<T>

- Preserves local FIFO order
  - If thread pushes and another thread pops two values, they come out in the same order that they went in
- Method push(const T&) places copy of item on back of queue
- Two kinds of pops
  - Blocking – pop(T&)
  - non-blocking – pop_if_present(T&)
- Method size() returns signed integer
  - If size() returns –n, it means n pops await corresponding pushes
- Method empty() returns size() == 0
  - Difference between pushes and pops
  - May return true if queue is empty, but there are pending pop()

Advice on using Concurrent Queue

A queue is inherently a bottleneck because it must maintain first-in first-out order.

A thread that is popping a value may have to wait idly until the value is pushed.

If a thread pushes an item and another thread pops it, the item must be moved to the other core.

Queue lets hot data grow cold in cache before it is consumed.

Use queue wisely, and consider rewriting your program using parallel_pipeline algorithm instead.

Concurrent Queue Container Example

```
#include <iostream>
#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);
      cout << "from queue: " << j << endl;
   }

   return 0;
}
```

Simple example to enqueue and print integers

Constructor for queue

Push items onto queue

While more things on queue

- Pop item off
- Print item

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 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

Concurrent Vector Container Example

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

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

Concurrent Hash Map Container

concurrent_hash_map<Key,T,HashCompare>

- Maps *Key* to element of type *T*
- You can define a class *HashCompare* with two methods
  - hash() maps Key to hashcode of type size_t
  - equal() returns true if two Keys are equal
- Enables concurrent count(), find(), insert(), and erase() operations
  - find() and insert() set "smart pointer" that acts as lock on item
    - **accessor** grants read-write access
    - **const_accessor** grants read-only access
  - lock released when smart pointer is destroyed

Concurrent Hash Map Container Example Key Insert

```
typedef concurrent_hash_map<string,int> myMap;
myMap table;
string newstring;
int place = 0;
…
  while (getNextString(&newString)) {
    myMap::accessor a;
    if (table.insert( a, newString ))  // new string inserted
       a->second = ++place;
  }
```

If insert() returns *true*, new string insertion

- Value is key’s place within sequence of strings from getNextString()

Otherwise, string has been previously seen

Concurrent Hash Map Container Example Key Find

```
myMap table;
string s1, s2;
int p1, p2;
…
{
    myMap::const_accessor a;  // read_lock
    myMap::const_accessor b;
    if (table.find(a,s1) && table.find(b,s2)) {  // find strings
        p1 = a->second; p2 = b->second;
        if (p1 < p2)
            cout << s1 << " came before " << s2 << endl;
        else
            cout << s2 << " came before " << s1 << endl;
        }
    else
        cout << "One or both strings not seen before" << endl;
}
```

If find() returns *true*, key was found within hash table

second contains the insertion number of the element.

Concurrent Hash Map vs Concurrent Unordered Map

Advantages :

- Concurrent insertion and traversal are allowed
- Interface is same as std::unordered_map

Disadvantages :

- There is no lock on items, but you can use atomic operations when you want to concurrently manipulate an item in the map (example : counters)
- Concurrent erasure is not allowed
- And... no order

Activity 4: Concurrent Container Lab

Use a hash table (concurrent_hash_map or concurrent_unordered_map) to keep track of the number of string occurrences.

Scalable Memory Allocators

Serial memory allocation can easily become a bottleneck in multithreaded applications

- Threads require mutual exclusion into shared heap

False sharing - threads accessing the same cache line

- Even accessing distinct locations, cache line can ping-pong

Intel® Threading Building Blocks 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`**
  - Offers both scalability and false sharing protection

Methods for `scalable_allocator`

```
#include <tbb/scalable_allocator.h>
template<typename T> class scalable_allocator;
```

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 );`

STL allocator functionality

- `T* A::allocate( size_type n, void* hint=0 )`
  - Allocate space for *n* values
- `void A::deallocate( T* p, size_t n )`
  - Deallocate *n* values from *p*
- `void A::construct( T* p, const T& value )`
- `void A::destroy( T* p )`

Scalable Allocators Example

```
#include <tbb/scalable_allocator.h>
typedef char _Elem;
typedef std::basic_string<_Elem,
                 std::char_traits<_Elem>,
                 tbb::scalable_allocator<_Elem>> MyString;

...

    int *p;
    MyString str1 = "qwertyuiopasdfghjkl";
    MyString str2 = "asdfghjklasdfghjkl";
    p = tbb::scalable_allocator<int>().allocate(24);
```

Intel Software

Activity 5: Memory Allocation Comparison

Do scalable memory exercise that first uses “new” and then asks user to replace with TBB scalable allocator

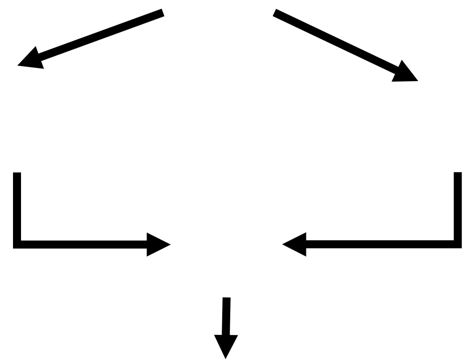
Flow Graph

Some applications best express dependencies as messages passed between nodes in a flow graph :

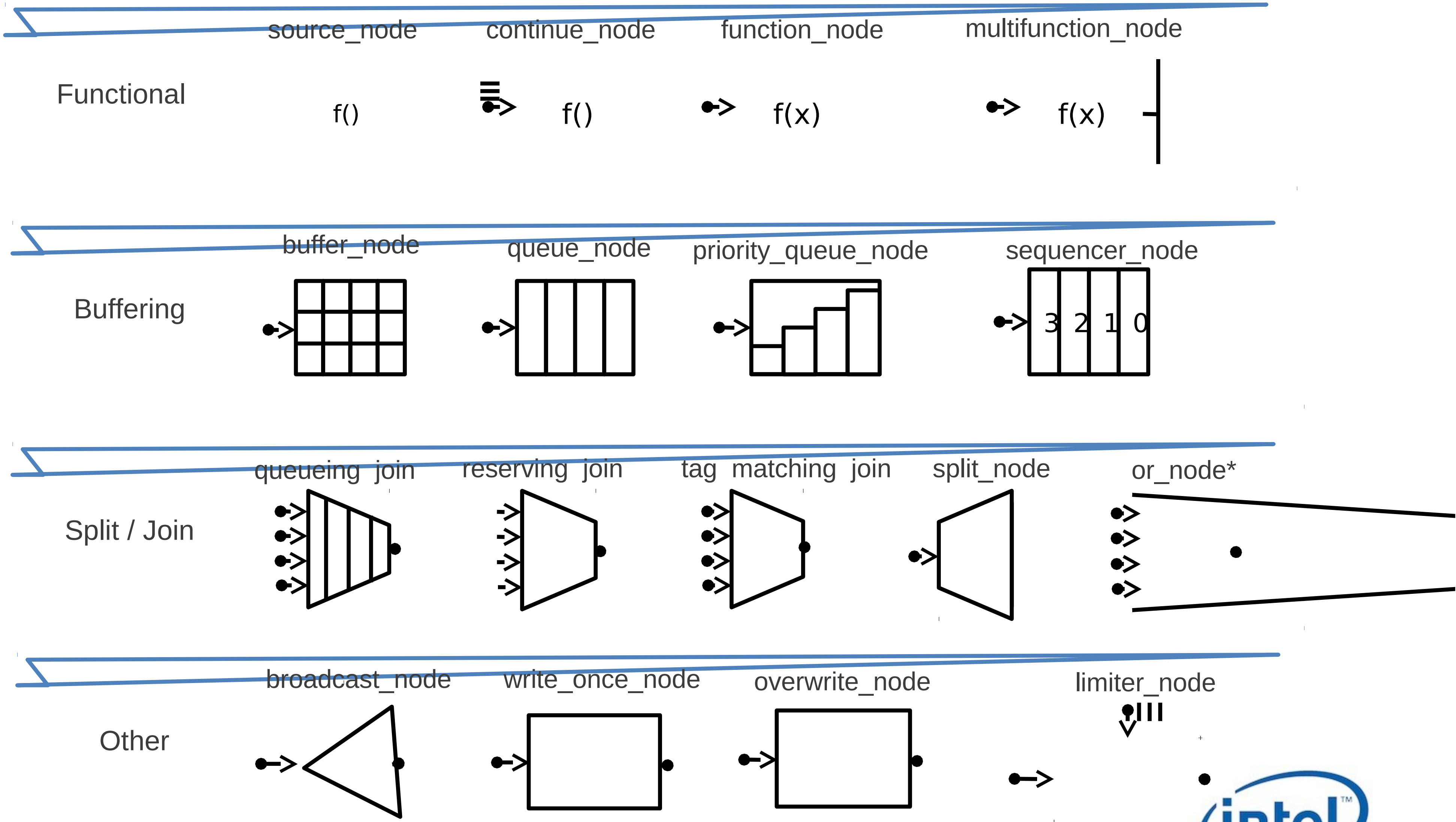
- Reactive applications that respond to events for asynchronous processing
- Task graph with complicated interrelationships
- Applications where nodes act like actors or agents that communicate by passing messages
- ...

Flow Graph

Graph object



Flow Graph Node Types



Software

Flow Graph – Hello World

```
#include <iostream>
#include <tbb/flow_graph.h>

using namespace std;
using namespace tbb::flow;

int main() {
        graph g;

        continue_node< continue_msg > h( g, []( const continue_msg & )
{ std::cout << "Hello "; } );
        continue_node< continue_msg > w( g, []( const continue_msg & )
{ std::cout << "Flow Graph World\n"; } );

        make_edge( h, w );

        h.try_put(continue_msg());
        g.wait_for_all();
}
```

Software

Activity 6 : Flow Graph

Flow graph is a really powerful feature but you must pay attention to the type of nodes you use.

Correct the proposed part of a feature detection flow graph : read the reference documentation on join_node and modify detection_join accordingly to avoid the current data race.

Synchronization Primitives

Parallel tasks must sometimes touch shared data

- When data updates might overlap, use mutual exclusion to avoid race

High-level generic abstraction for HW atomic operations

- Atomically protect update of single variable

Synchronization Primitives

Critical regions of code are protected by scoped locks

- The range of the lock is determined by its lifetime (scope)
- Leaving lock scope calls the destructor, making it exception safe
- Minimizing lock lifetime avoids possible contention
- Several mutex behaviors are available
  - Spin vs. queued
    - “are we there yet” vs. “wake me when we get there”
  - Writer vs. reader/writer (supports multiple readers/single writer)
  - Scoped wrapper of native mutual exclusion function

Atomic Execution

atomic<T>

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

‘= x’ and ‘x = ’	read/write value of x
x.fetch_and_store (y)	z = x, y = x, return z
x.fetch_and_add (y)	z = x, x += y, return z
x.compare_and_swap (y,p)	z = x, if (x==p) x=y; return z

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

i is incremented by 2 and the value of **z** will be the value that was previously stored in **i**.

intel Software

Mutex Concepts

Mutexes are C++ objects based on scoped locking pattern

Combined with locks, provide mutual exclusion

M()	Construct unlocked mutex
~M()	Destroy unlocked mutex
typename M::scoped_lock	Corresponding scoped_lock type
M::scoped_lock ()	Construct lock w/out acquiring a mutex
M::scoped_lock (M&)	Construct lock and acquire lock on mutex
M::~scoped_lock ()	Release lock if acquired
M::scoped_lock::acquire (M&)	Acquire lock on mutex
M::scoped_lock::release ()	Release lock

Mutex Flavors

Fair: A *fair mutex* allows threads to execute the critical region in the order in which the threads arrive. However, *unfair mutexes* allow threads that are already running to execute the critical region first instead of the thread that is next in line, so they are faster.

Reentrant: A *reentrant mutex* allows a thread that is holding a lock on the mutex to acquire another lock on the mutex. This is useful in the case of recursive algorithms in which the same function is called repeatedly and you need to lock the function each time it is called. However, additional locks also add to the overheads.

Spin: Mutexes can keep a thread busy by making it spin in user space or sleep while a thread is waiting for another thread to release the mutex. Making a thread sleep while waiting consumes CPU cycles, so if a thread need to wait only for a short duration, the spin version is better.

Mutex Flavors

spin_mutex

- Non-reentrant, unfair, spins in the user space
- VERY FAST in lightly contended situations; use if you need to protect very few instructions

queuing_mutex

- Non-reentrant, fair, spins in the user space
- Use Queuing_Mutex when scalability and fairness are important

queuing_rw_mutex

- Non-reentrant, fair, spins in the user space

spin_rw_mutex

- Non-reentrant, fair, spins in the user space
- Use ReaderWriterMutex to allow non-blocking read for multiple threads

mutex

- Wrapper for OS sync: CRITICAL_SECTION for Windows*, pthread_mutex on Linux*

recursive_mutex

- Like mutex, but reentrant

Reader-Writer Lock Example

```
#include <tbb/spin_rw_mutex.h>
using namespace tbb;

spin_rw_mutex MyMutex;

int foo (){
    /* Construction of ‘lock’ acquires ‘MyMutex’ */
    spin_rw_mutex::scoped_lock lock (MyMutex, /*is_writer*/ false);

    read_shared_data (data);

    if (!lock.upgrade_to_writer ()) {
       /* lock was released to upgrade;
          may be unsafe to access data, recheck status before use */
    }
    else {
       /* lock was not released; no other writer was given access */
    }

    return 0;
    /* Destructor of ‘lock’ releases ‘MyMutex’ */
}
```

One last question…

How do I know how many threads are available?

Do not ask!

- Not even the scheduler knows how many threads really are available
- There may be other processes running on the machine
- Routine may be nested inside other parallel routines

Focus on dividing your program into tasks of sufficient size

- Task should be big enough to amortize scheduler overhead
- Choose decompositions with good depth-first cache locality and potential breadth-first parallelism

Let the scheduler do the mapping

Resources

Website

- http://threadingbuildingblocks.org/

- Latest Open Source Documentation :
  http://threadingbuildingblocks.org/ver.php?fid=91

Forum

- http://software.intel.com/en-us/forums/intel-threading-building-blocks/

Blogs

- http://software.intel.com/en-us/blogs/tag/tbb/

intel®

Leap ahead™

License Creative Commons - By 2.5