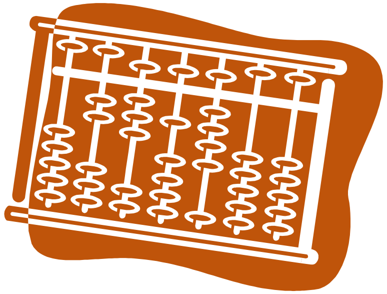
Parallel Programming Principle and Practice

Lecture 7



Threads programming with TBB

Outline

- Intel Threading Building Blocks
- Task-based programming
- Task Scheduler
- Scalable Memory Allocators
- Concurrent Containers
- Synchronization Primitives

Ways to Improve Naïve Implementation

- Programming with OS Threads can get complicated and error-prone, even for the pattern as simple as for-loop

Problems with Naïve Implementation	What You Could Do to Improve It
Works with *fixed number of threads*	Implement a function which determines the ideal number of worker threads
The implementation is *not portable*	Implement wrapper functions with code specific to each supported OS
The solution is *not re-usable*	Abstract the iteration space and re-write all the loops to comply with it
Potentially *poor performance* due to work-load imbalance	Implement thread-pool and use heuristics to balance the work-load between worker threads
The solution is *not composable*	Well…continue adding more code…doing testing…and tuning…

Task-basked Programming
A Better Approach to Parallelism

- Portable task-based technologies
  - Intel® Threading Building Blocks (Intel® TBB)
    - lets you easily write parallel C++ programs that take full advantage of multicore performance, that are portable and composable, and that have future-proof scalability.
    - C++ template library: general task-based programming, concurrent data containers, and more …

Key Feature

- It is a *template library* intended to ease parallel programming for C++ developers
  - Relies on generic programming to deliver high performance parallel algorithms with broad applicability
- It provides a *high-level abstraction* for parallelism
- It facilitates scalable performance
  - Strives for efficient use of cache, and balances load
  - Portable across Linux*, Mac OS*, Windows*, and Solaris*
- *Can be used in concert with other packages such as native threads and OpenMP (fighting for thread, tbb, openmp)*
- Open source and licensed versions available

Implement "parallel ideals" with Templates and Language Features

Typical Serial Program	Ideal Parallel Program	Issues
Algorithms	Parallel Algorithms	Require many code changes when developed from scratch: often it takes a threading expert to get it right
Data Structures	Thread-safe and scalable Data Structures	Serial data structures usually require global locks to make operations thread-safe
Dependencies	- Minimum of dependencies - Efficient use of synchronization primitives or thread local storage	Too many dependencies → expensive synchronization → poor parallel performance
Memory Management	Scalable Memory Manager	Standard memory allocator is often inefficient in multi-threaded app

Task-based Programming Advantages

	OS Threads	Intel® Cilk™ Plus Intel® Threading Building Blocks
Forward-scaling	*Takes a threading expert* to implement a scalable solution	Allow thinking at higher level and produce implementations *independent of number of CPUs*
Portability	*Non-portable*, requires extra coding, maintenance, and testing	*Portable* across many platforms
Flexibility	Requires *extra effort* to implement reusable solution	*Broadly applicable* by design
Performance	Requires a threading expert and special knowledge to get it right	Designed for *high performance*
Composability	Cross-component coordination is required (added coding, testing, and tuning)	Support *nested parallelism* and can be used together
Conclusion:	An efficient solution using OS threads requires expertise and leads to a significant re-design	Task-based solution often can speed up your app with a minimal code changes

Implementing Common Parallel Performance Patterns

Parallel Program Components	Intel® Parallel Building Blocks
Parallel Algorithms	Intel® Cilk™ Plus and Intel® Threading Bulling Blocks (Intel® TBB) parallel **loops**, parallel **functions**, parallel **recursion**, parallel **pipeline**
Thread-safe and Scalable Data Structures	Intel TBB **concurrent containers**
Dependencies	Intel TBB **flow graph**
Thread-Local Storage	Intel Cilk Plus **reducers** Intel TBB **thread-local storage**
Synchronization Primitives	Intel TBB exception-safe **locks, condition variables**, and **atomics**
Scalable Memory Manager	Intel TBB **scalable memory allocator** and **false-sharing free allocator**

Intel® TBB online

www.threadingbuildingblocks.org

Downloads, active users forum, developers' blogs, documentation

News and announcements

Open Source License information

Code samples, FAQ

limitation

- TBB is not intended for
  - I/O bound processing
  - Real-time processing
- General limitations
  - Direct use only from C++
  - Distributed memory not supported (target is desktop)
  - Requires more work than sprinkling in pragmas, for example OpenMP

Outline

- Intel Threading Building Blocks
- Task-based programming
- Task Scheduler
- Scalable Memory Allocators
- Concurrent Containers
- Synchronization Primitives

Task-based programming

- Tasks are light-weight entities at user-level
  - TBB parallel algorithms map tasks onto threads automatically
  - Task scheduler manages the thread pool
    - Scheduler is *unfair* to favor tasks that <u>have been most recent in the cache</u>

Why unfair?

- Oversubscription and undersubscription of core resources is prevented by task-stealing technique of TBB scheduler

Generic Parallel Algorithms

Loop parallelization

- **parallel_for** and **parallel_reduce** Load balanced parallel execution of fixed number of independent loop iterations
- **parallel_scan** Template function that computes parallel prefix (y[i] = y[i-1] op x[i])

Parallel Algorithms for Streams

- **parallel_do** Use for unstructured stream or pile of work; Can add additional work to pile while running
- **parallel_for_each** parallel_do without an additional work feeder

Generic Parallel Algorithms

Parallel Algorithms for Streams

- **pipeline / parallel_pipeline**
  - Linear pipeline of stages - you specify maximum number of items that can be in flight
  - Each stage can be parallel or serial in-order or serial out-of-order. Stage (filter) can also be thread-bound
  - Uses cache efficiently: Each worker thread flies an item through as many stages as possible; Biases towards finishing old items before tackling new ones

Others

- **parallel_invoke** Parallel execution of a number of user-specified functions
- **parallel_sort** Comparison sort with an average time complexity O(N Log(N)); When worker threads are available parallel_sort creates subtasks that may be executed concurrently

The parallel_for Template

```
template <typename Range, typename Body>
void parallel_for(const Range& range, const Body &body);
```

- Requires definition of
  - A range type to iterate over
    - Must define a copy constructor and a destructor
    - Defines is_empty ()
    - Defines is_divisible ()
    - Defines a splitting constructor, R(R &r, split)
  - A body type that operates on the range (or a subrange)
    - Must define a copy constructor and a destructor
    - Defines operator()

Body is Generic

- Requirements for parallel_for Body

Body::Body(const Body&)	Copy constructor
Body::~Body()	Destructor
void Body::operator() (Range& *subrange*) const	Apply the body to *subrange*.

- 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

- Requirements for parallel_for Range

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

- Library provides predefined ranges
  - blocked_range and blocked_range2d
- You can define your own ranges

An Example using parallel_for (1 of 4)

- Independent iterations and fixed/known bounds
- Sequential code starting point

```
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 (2 of 4)

- Include and initialize the library

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

```
#include “tbb/task_scheduler_init.h”
#include “tbb/blocked_range.h”
#include “tbb/parallel_for.h”
using namespace tbb;

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

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

An Example using parallel_for (3 of 4)

- Use the parallel_for algorithm

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

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

```
class ChangeArrayBody {
    float *array;
  public:
    ChangeArrayBody (float *a): array(a) {}
    void operator() ( const blocked_range <int>& r ) const {
            for (int i=r.begin(); i != r.end(); i++) {
                  array[i] *= 2;
            }
    }
};
void  parallel_change_array(float *array, int M) {
    parallel_for (blocked_range <int>(0,M),
                  ChangeArrayBody(array), auto_partitioner() ) ;
}
```

An Example using parallel_for (4 of 4)

- Use the parallel_for algorithm

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

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

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

Parallel algorithm usage example

```cpp
#include "tbb/blocked_range.h"
#include "tbb/parallel_for.h"
using namespace tbb;

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

void ChangeArrayParallel (int* a, int n )
{
   parallel_for (blocked_range<int>(0, n), ChangeArrayBody(a));
}

int main (){
   int A[N];
   // initialize array here…
   ChangeArrayParallel (A, N);
   return 0;
}
```

ChangeArrayBody class defines a for-loop body for parallel_for

blocked_range – TBB template representing 1D iteration space

As usual with C++ function objects the main work is done inside operator()

A call to a template function parallel_for<Range, Body>: with arguments
Range → blocked_range
Body → ChangeArray

Outline

- Intel Threading Building Blocks
- Task-based programming
- Task Scheduler
- Scalable Memory Allocators
- Concurrent Containers
- Synchronization Primitives

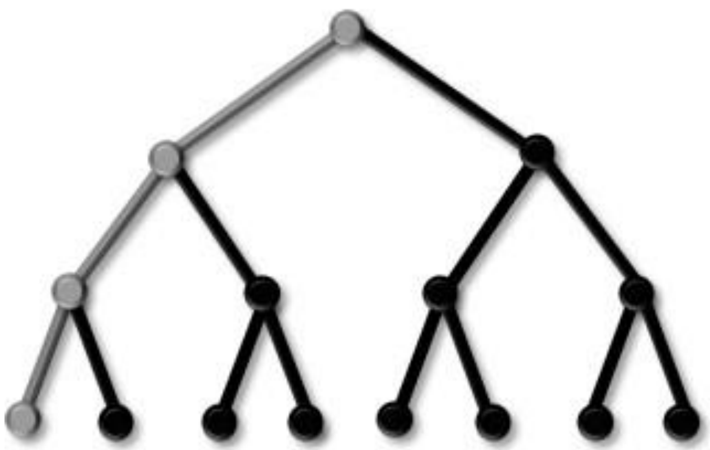
Task Scheduler

- Task scheduler is the engine driving Intel® Threading Building Blocks
  - Manages thread pool, hiding complexity of native thread management
  - Maps logical tasks to threads
- Parallel algorithms are based on task scheduler interface
- Task scheduler is designed to address common performance issues of parallel programming with native 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

Two Execution Orders

Depth First
(stack)

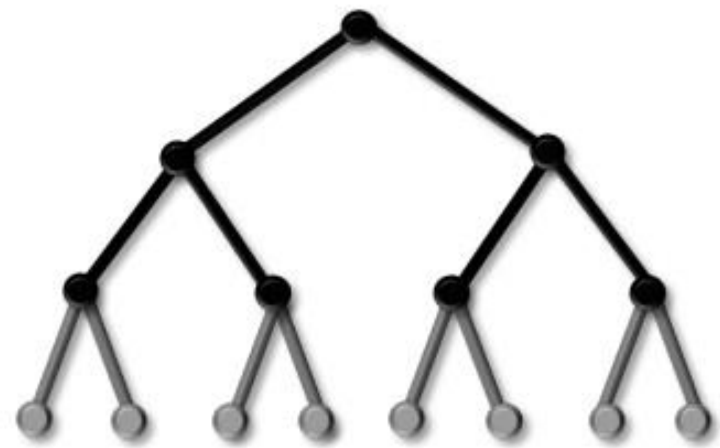


Small space

Excellent cache locality

No parallelism

Breadth First
(queue)

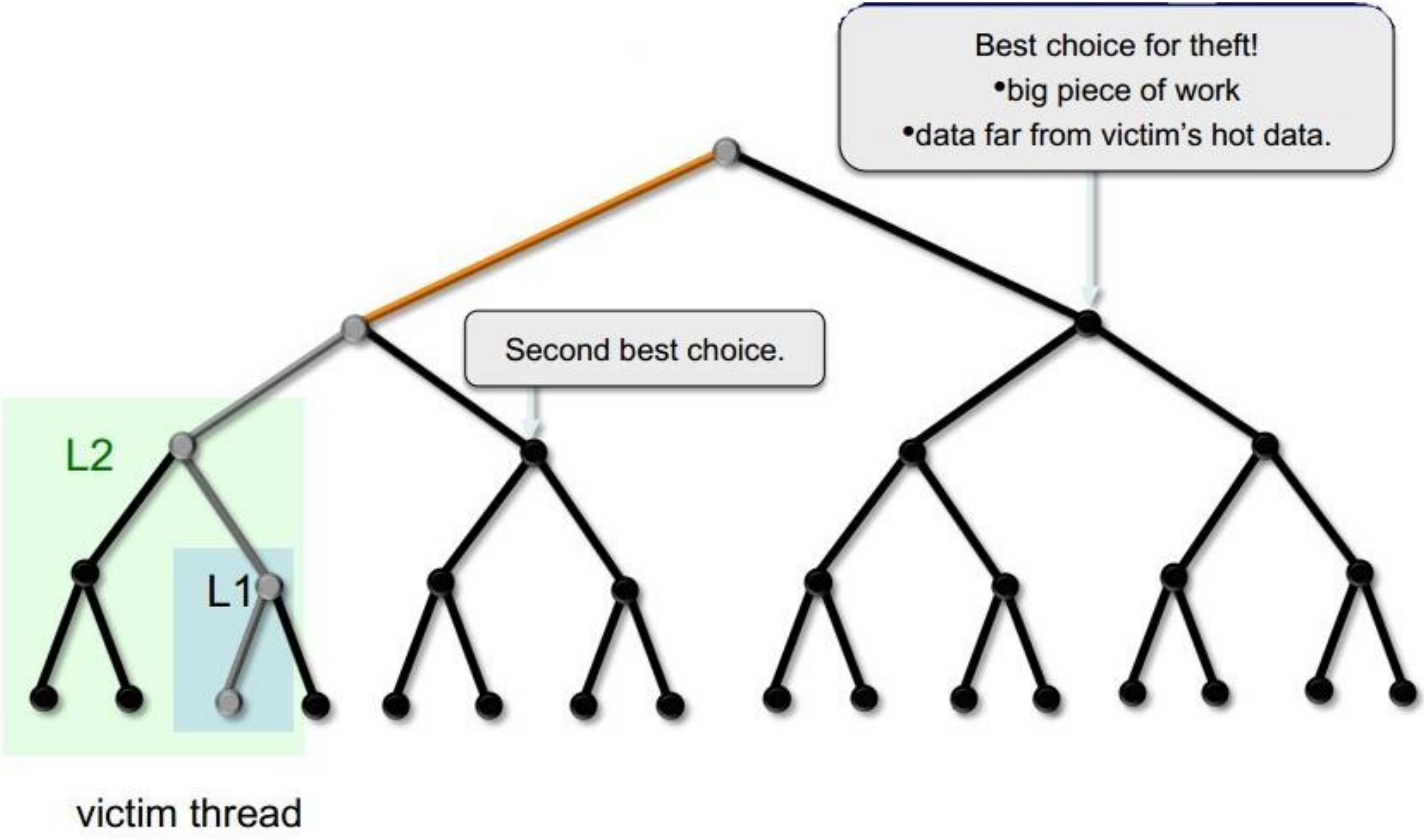


Large space

Poor cache locality

Maximum parallelism

Work Depth First; Steal Breadth First



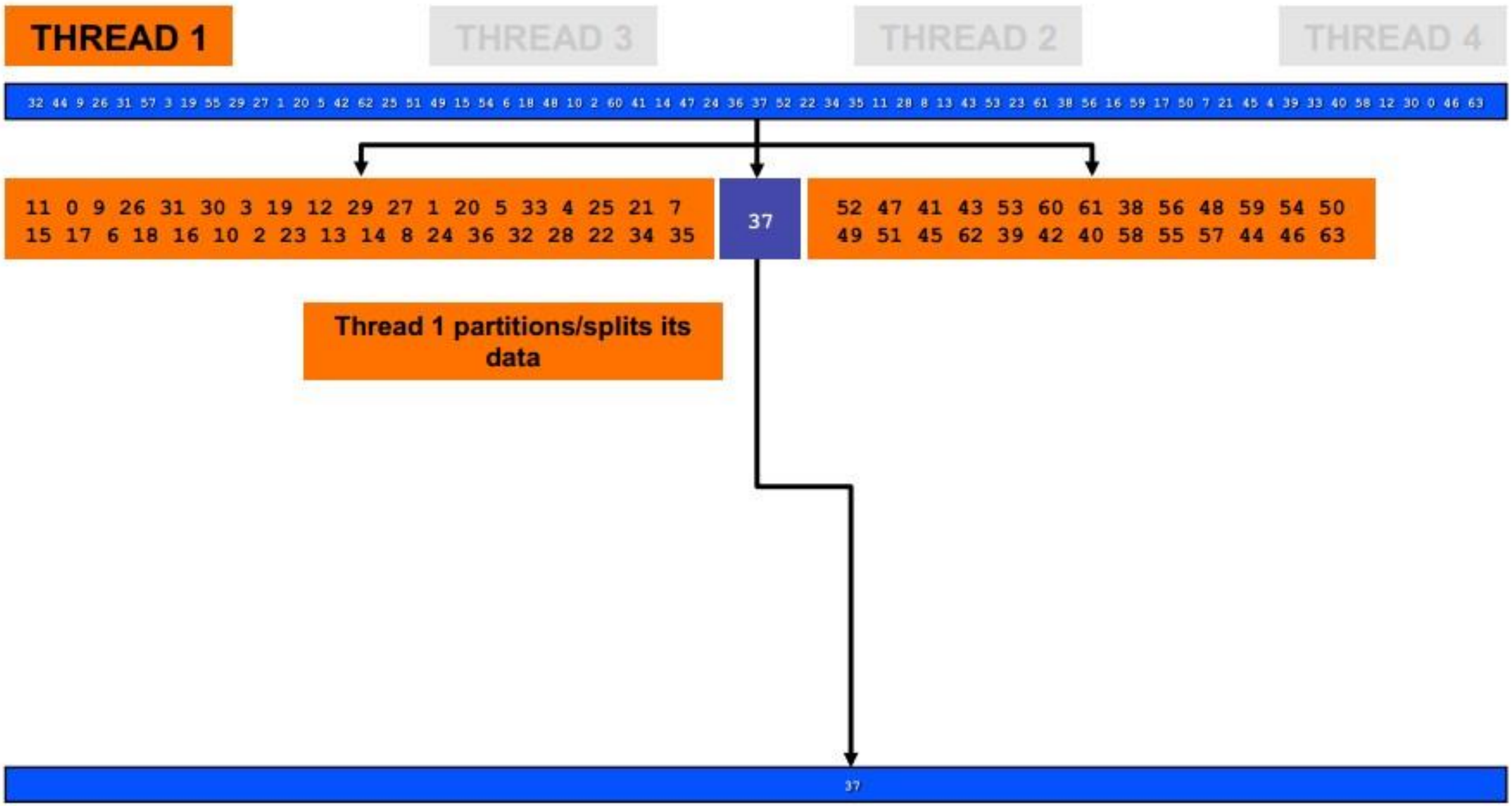
Another example: Quicksort – Step 1

THREAD 1

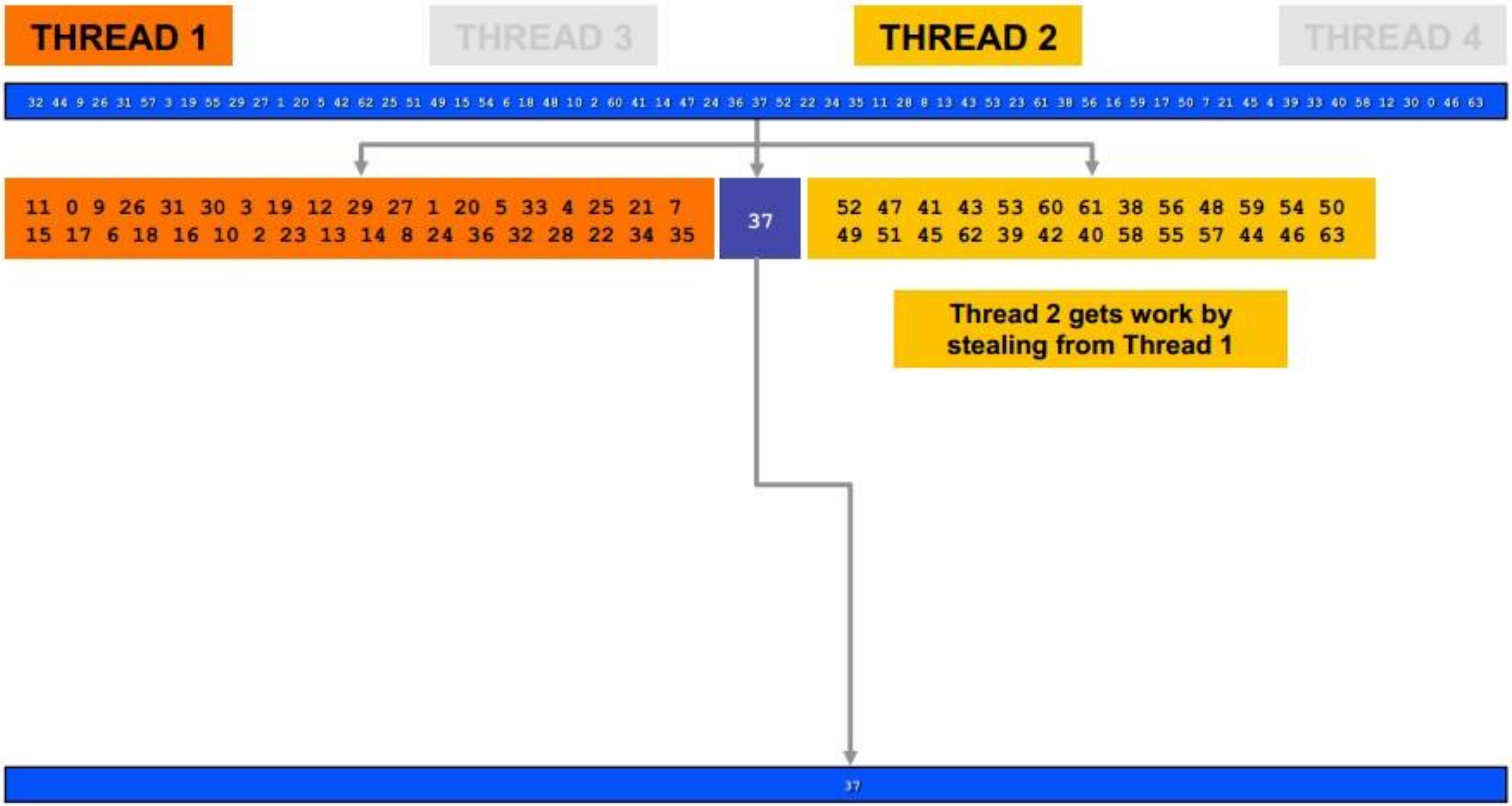
32 44 9 26 31 57 3 19 55 29 27 1 20 5 42 62 25 51 49 15 54 6 18 48 10 2 60 41 14 47 24 36 37 52 22 34 35 11 28 8 13 43 53 23 61 38 56 16 59 17 50 7 21 45 4 39 33 40 58 12 30 0 46 63

Thread 1 starts with the initial data

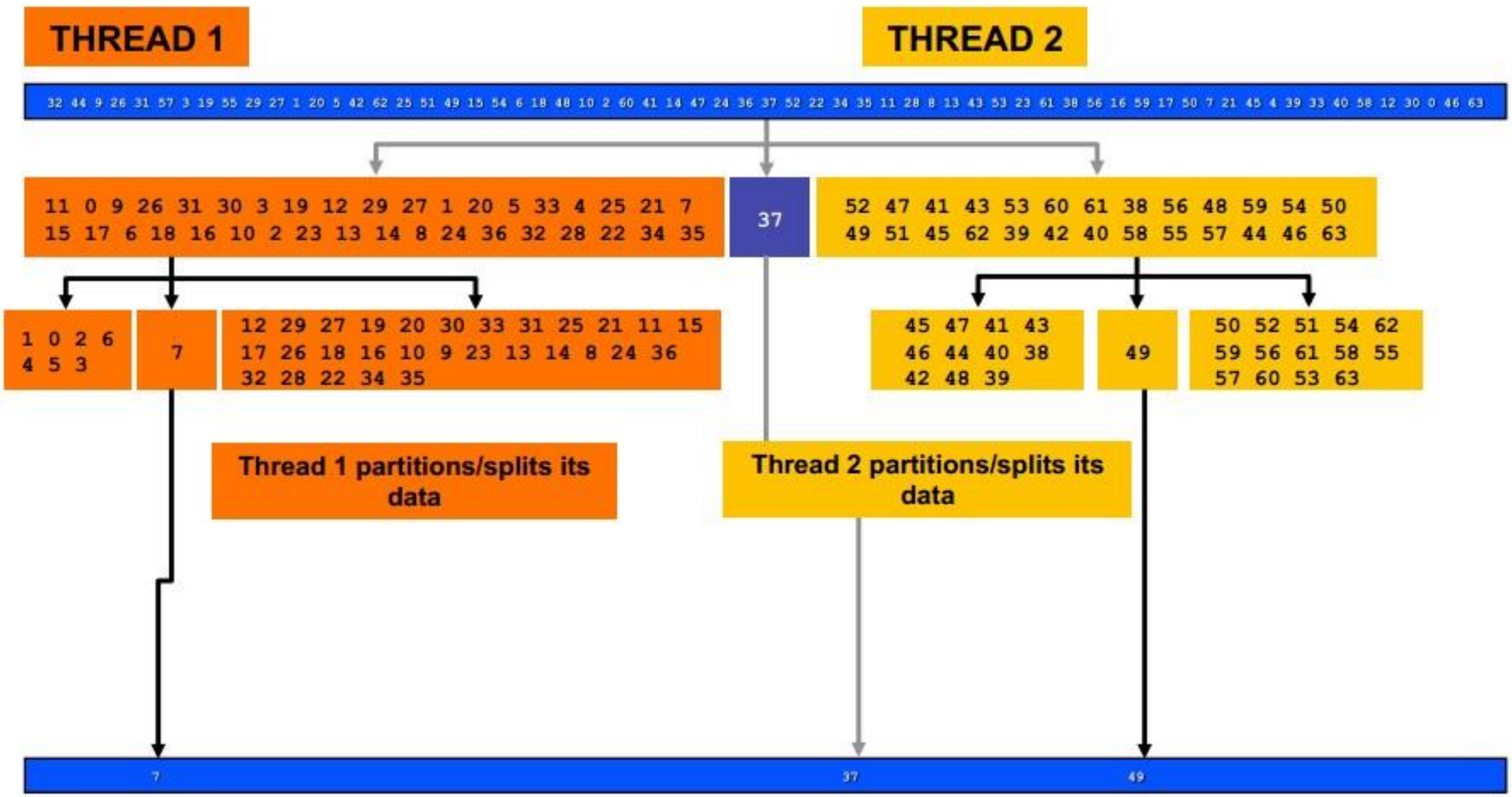
Quicksort – Step 2



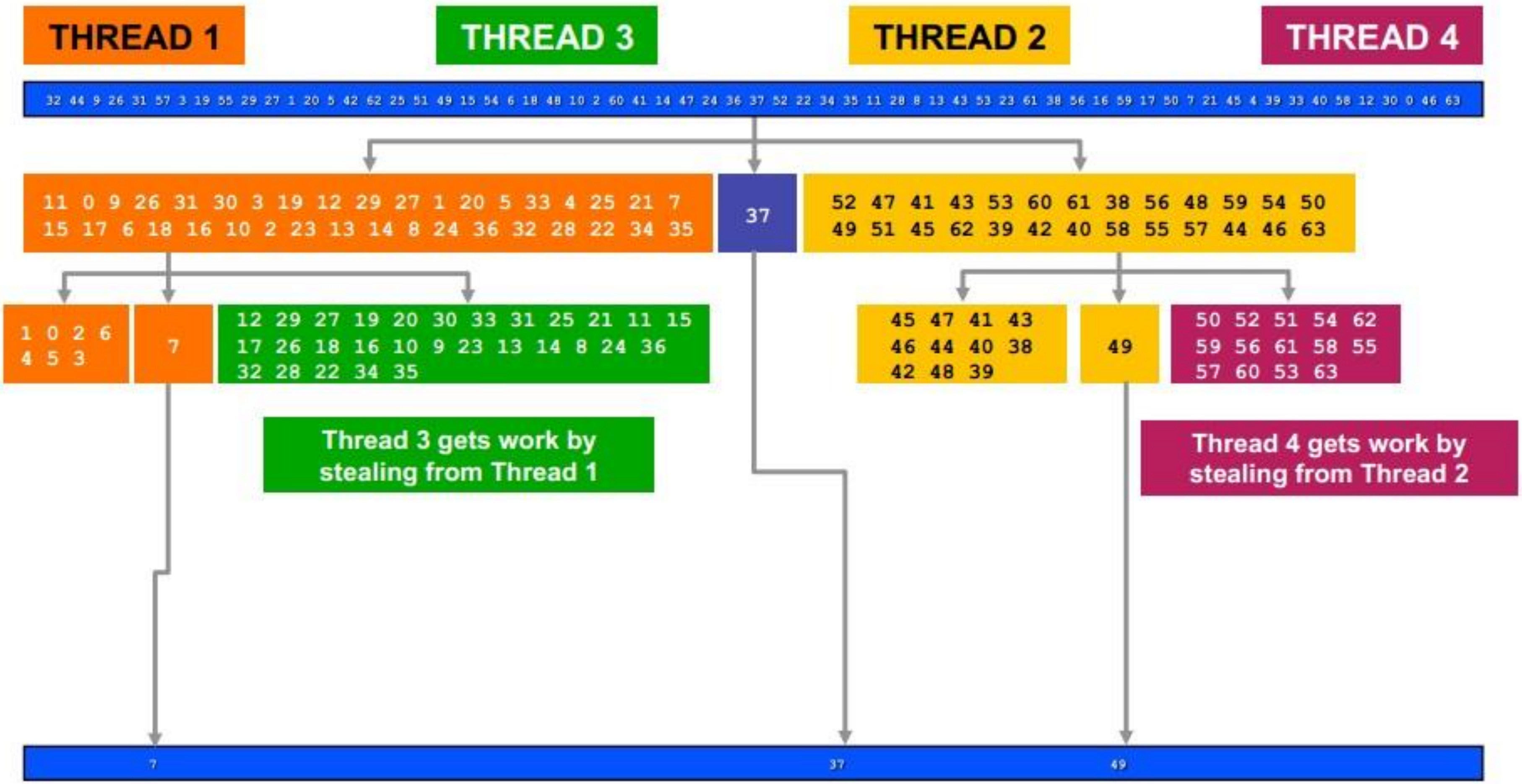
Quicksort – Step 2



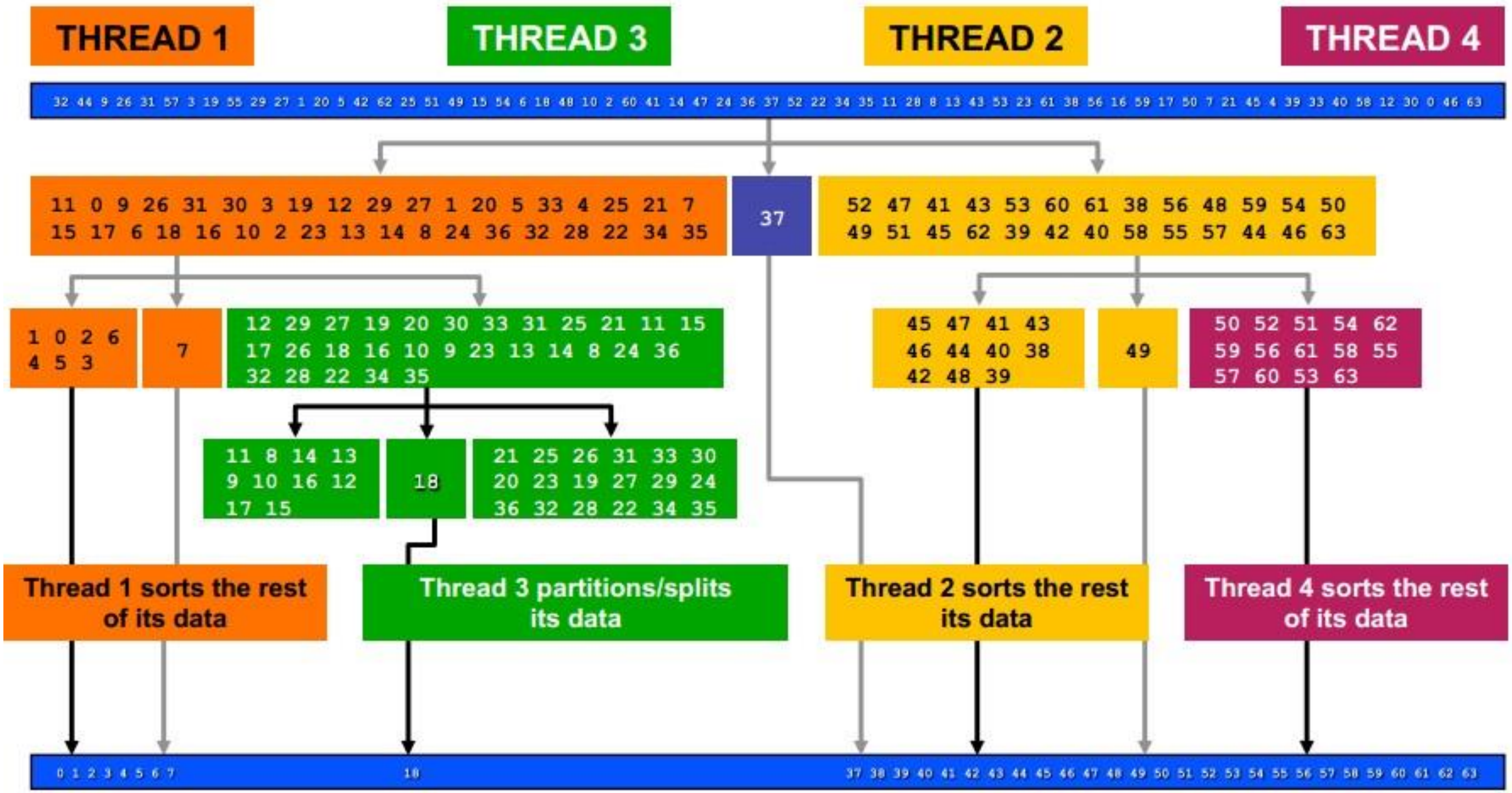
Quicksort – Step 3



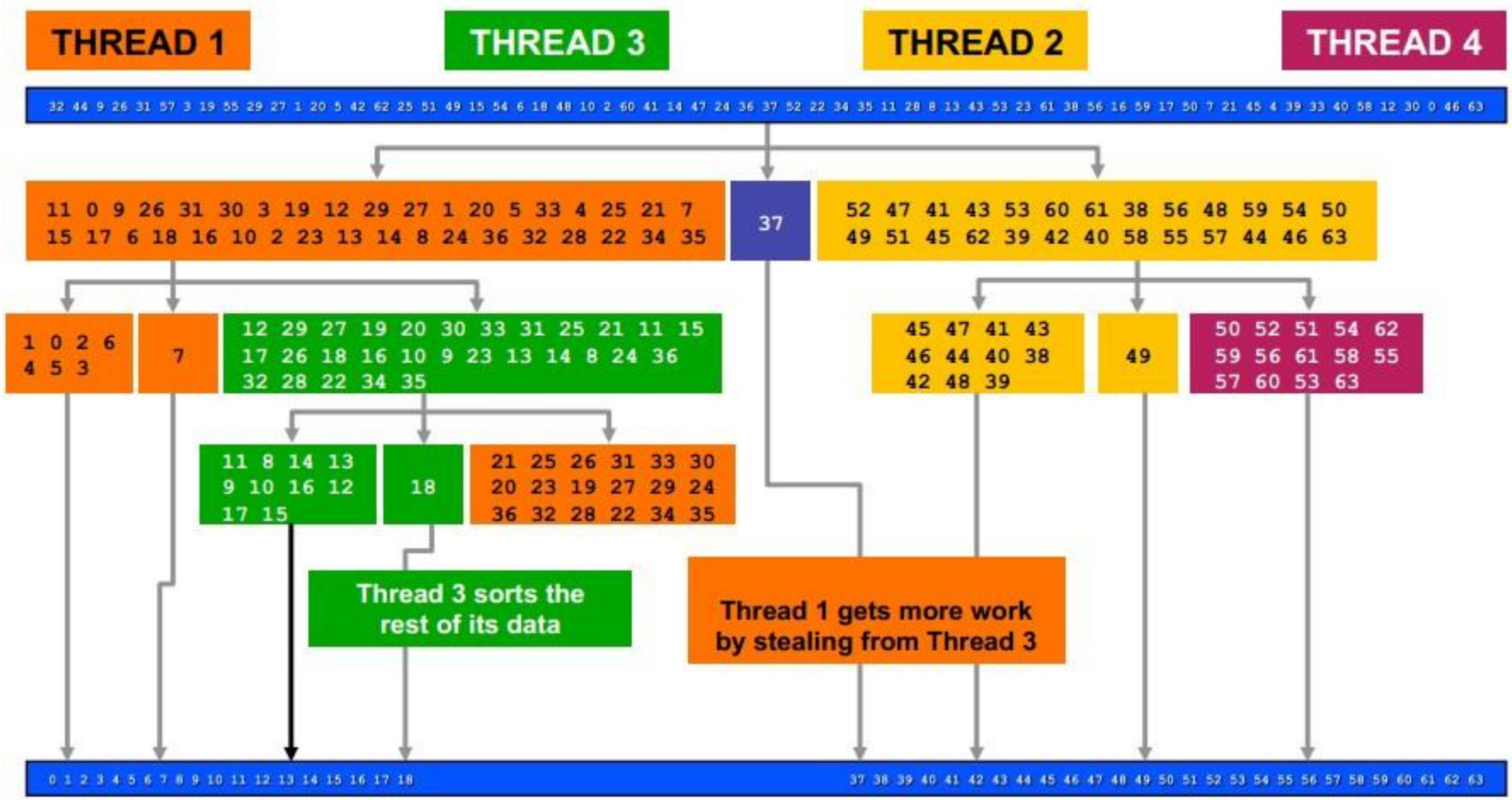
Quicksort – Step 3



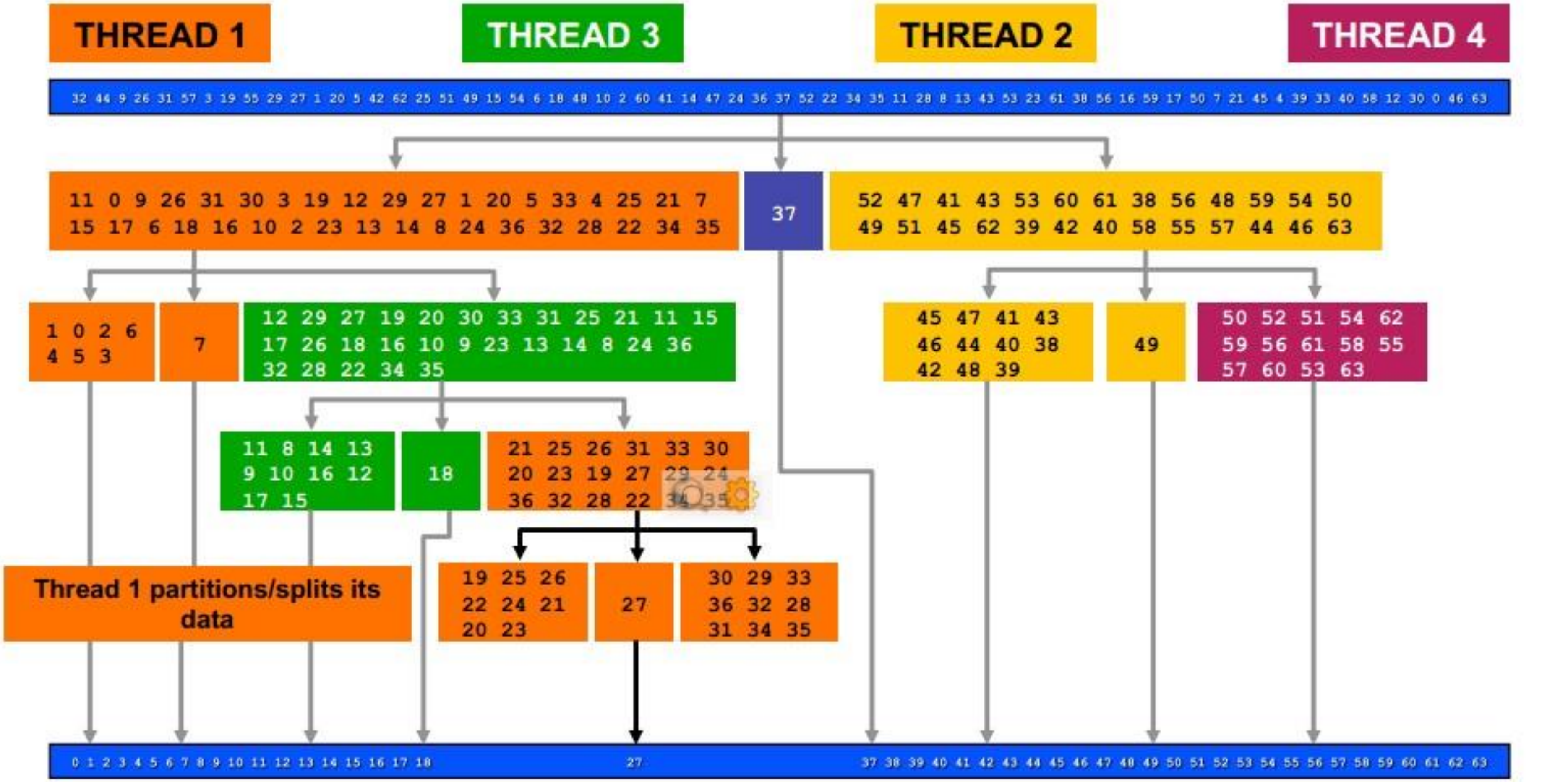
Quicksort – Step 4



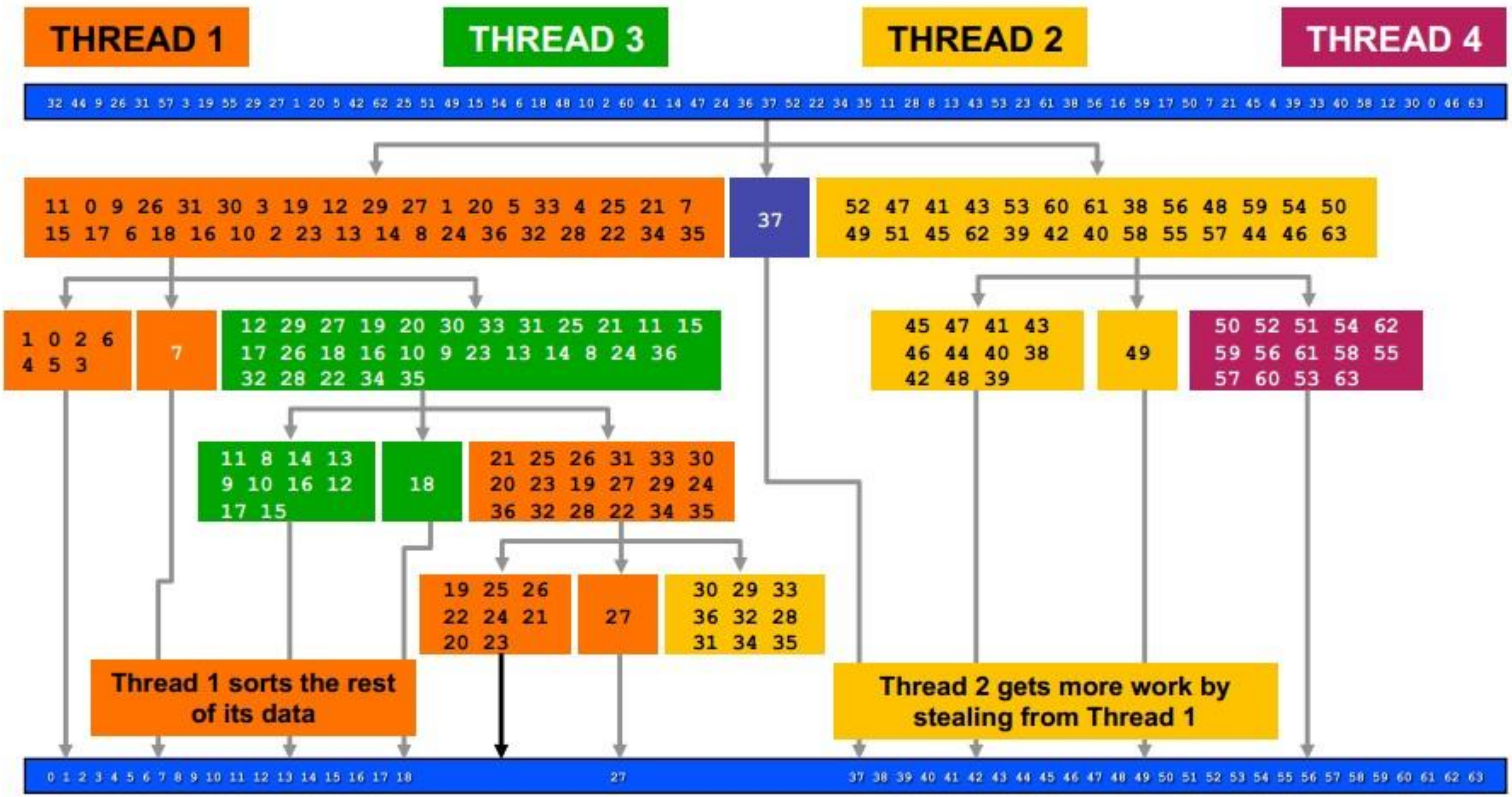
Quicksort – Step 5



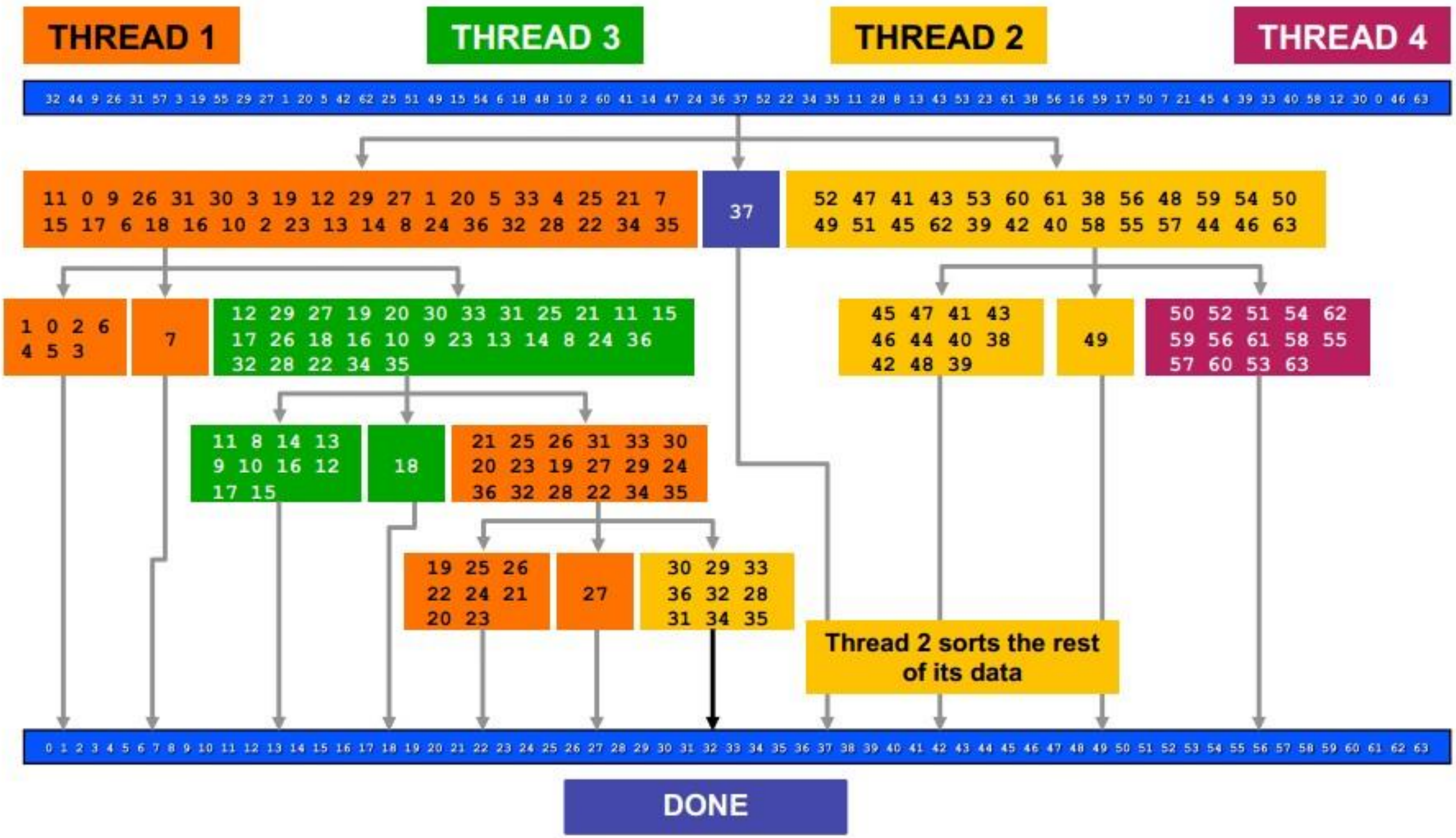
Quicksort – Step 6



Quicksort – Step 6



Quicksort – Step 7



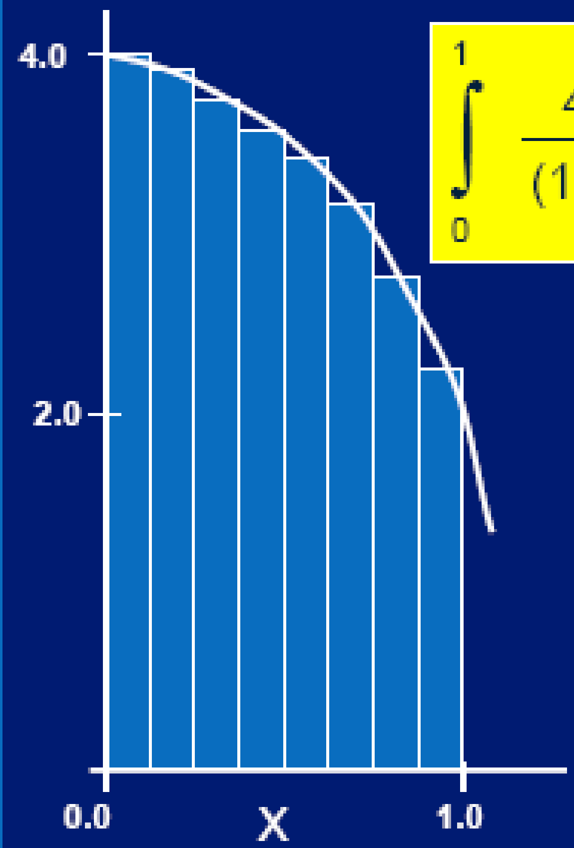
The parallel_reduce Template

```
template <typename Range, typename Body>
void parallel_reduce (const Range& range, Body &body);
```

- Requirements for parallel_reduce Body

Body::Body(const Body&, split)	Splitting constructor
Body::~Body()	Destructor
void Body::operator() (Range& *subrange*) const	Accumulate results from *subrange*
void Body::join(Body& *rhs*);	Merge result of *rhs* into the result of this.

Numerical Integration Example (1 of 3)



$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx$$

```
static long num_steps=100000;
double step, pi;

void main(int argc, char*
argv[])
{   int i;
    double x, sum = 0.0;

    step = 1.0/(double) num_steps;
    for (i=0; i< num_steps; i++){
       x = (i+0.5)*step;
       sum += 4.0/(1.0 + x*x);
    }
    pi = step * sum;
    printf("Pi = %f\n",pi);
}
```

parallel_reduce Example (2 of 3)

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

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

using namespace tbb;

int main(int argc, char* argv[])
{
  double pi;
  double width = 1./(double)num_steps;
  MyPi step((double *const)&width);
  task_scheduler_init init;

  parallel_reduce(blocked_range<size_t>(0,num_steps), step,
                                                   auto_partitioner() );

  pi = step.sum*width;

  printf("The value of PI is %15.12f\n",pi);
  return 0;
}
```

parallel_reduce Example (3 of 3)

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

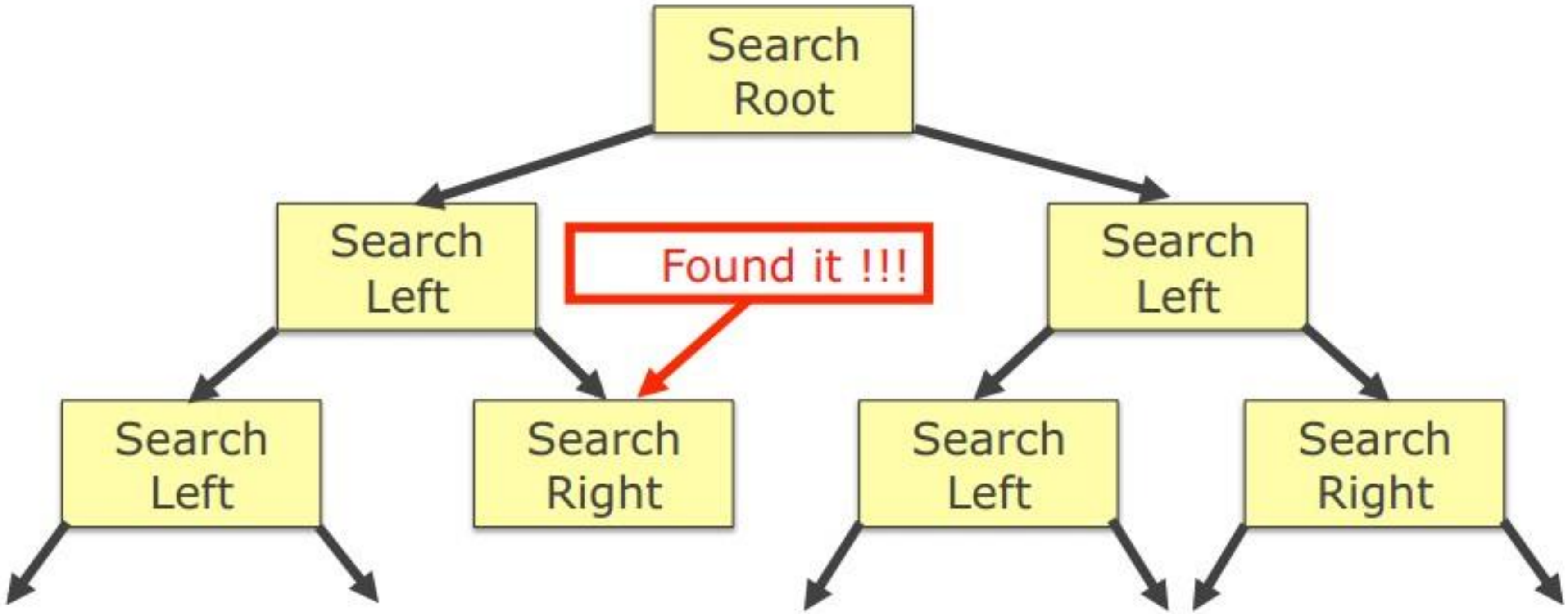
```
class MyPi {
  double *const my_step;
public:
  double sum;
  void operator()( const blocked_range<size_t>& r ) {
    double step = *my_step;
    double x;
    for (size_t i=r.begin(); i!=r.end(); ++i)
    {
       x = (i + .5)*step;
       sum += 4.0/(1.+ x*x);
    }
  }

  MyPi( MyPi& x, split ) : my_step(x.my_step), sum(0) {}

  void join( const MyPi& y ) {sum += y.sum;}

  MyPi(double *const step) : my_step(step), sum(0) {}

};
```

accumulate results

join

Task cancellation avoids unneeded work

There is a whole class of application that can benefit from ability to cancel work early



Task cancellation example

```
const int problemSize = N;

int main() {
    vector<int> intVec(problemSize);
    const int valToFind = K;
    int valIdx = -1;

    parallel_for( blocked_range<int>(0, problemSize),
        [&](const blocked_range<int>& r) {
            for( int i = r.begin(); i < r.end(); ++i ) {
                if ( intVec[i] == valToFind ) {
                    tbb::task::self().cancel_group_execution();
                }
            }
        });

    return 0;
}
```

When the value is found the task cancels itself and all the other tasks in the same "group" (by default these are all of the tasks of the same algorithm)

Uncaught exceptions cancel task execution

```
int main() {

   try {
      parallel_for( blocked_range<int>(0, N),
         [&](const blocked_range<int>& r) {
            ...
            for(int i = r.begin(); i != r.end(); ++i) {
               if (data[i] == bad_value)
                  throw std::logic_error("Bad value in list");
            }
            ...
      });
   } catch (tbb::captured_exception& e) {
      cout << e.name() << " with description: " << e.what() << endl;
   }

   return 0;
}
```

An exception thrown from inside the task does not need to be caught in the same task. It will cancel task group execution and can be caught from outside the algorithm

A **tbb::captured_task** can be handled in the catch block

Outline

- Intel Threading Building Blocks
- Task-based programming
- Task Scheduler
- Scalable Memory Allocators
- Concurrent Containers
- Synchronization Primitives

False sharing

- 在多核架构中，每个CPU都有自己的Cache，如果一个内存中的变量在每个cache都存在的话，就需要保证各个cache中变量的一致性
- Intel多核架构实现Cache一致性是采用的MESI (Modified/Exclusive/Shared/Invalid) 协议
- 缓存系统中是以缓存行（cache line）为单位存储的。缓存行是2的整数幂个连续字节，一般为32-256个字节
- 最常见的缓存行大小是64个字节

False sharing

- 当多线程修改互相独立的变量时，如果这些变量共享同一个缓存行，就会无意中影响彼此的性能，这就是伪共享
- 缓存行上的写竞争是运行在SMP系统中并行线程实现可伸缩性最重要的限制因素。有人将伪共享描述成无声的性能杀手，因为从代码中很难看清楚是否会出现伪共享
- 为了让可伸缩性与线程数呈线性关系，就必须确保不会有两个线程往同一个变量或缓存行中写

False sharing

- 共享必须满足2个条件：
  - 1 .数据都在一个cache line
  - 2 多CPU同时访问
- 两个线程写同一个变量可以在代码中发现。为了确定互相独立的变量是否共享了同一个缓存行，就需要了解内存布局，或找个工具告诉我们
- Intel VTune就是这样一个分析工具

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

Outline

- Intel Threading Building Blocks
- Task-based programming
- Task Scheduler
- Scalable Memory Allocators
- Concurrent Containers
- Synchronization Primitives

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
  - Can be used with the library, OpenMP, or native threads

Concurrent Containers Key Features

concurrent_hash_map <Key,T,Hasher,Allocator>

- Models hash table of std::pair <const *Key*, *T*> elements
- Maps *Key* to element of type *T*
- User defines *Hasher* to specify how keys are hashed and compared
- Defaults: Allocator=tbb::tbb_allocator

concurrent_unordered_map<Key,T,Hasher,Equality,Allocator>

- Permits concurrent traversal and insertion (no concurrent erasure)
- Requires no visible locking, looks similar to STL interfaces
- Defaults: Hasher=tbb::tbb_hash, Equality=std::equal_to, Allocator=tbb::tbb_allocator

concurrent_vector <T, Allocator>

- Dynamically growable array of T: grow_by and grow_to_atleast
- cache_aligned_allocator is a default allocator

concurrent_queue <T, Allocator>

- For single threaded run concurrent_queue supports regular "first-in-first-out" ordering
- If one thread pushes two values and the other thread pops those two values they will come out in the order as they were pushed
- cache_aligned_allocator is a default allocator

concurrent_bounded_queue <T, Allocator>

- Similar to concurrent_queue with a difference that it allows specifying capacity. Once the capacity is reached 'push' will wait until other elements will be popped before it can continue.

Outline

- Intel Threading Building Blocks
- Task-based programming
- Task Scheduler
- Scalable Memory Allocators
- Concurrent Containers
- Synchronization Primitives

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
- 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

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, x = y, 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);
```

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

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

References

- The content expressed in this chapter is come from
  - berkeley university open course (http://parlab.eecs.berkeley.edu/2010bootcampagenda, Shared Memory Programming with TBB, Michael Wrinn)
  - http://software.intel.com/en-us/courseware
  - IDF2012：Task Parallel Evolution and Revolution – Intel Cilk Plus and Intel Threading Building Blocks