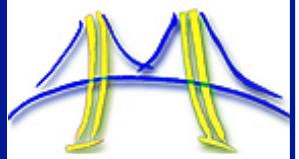
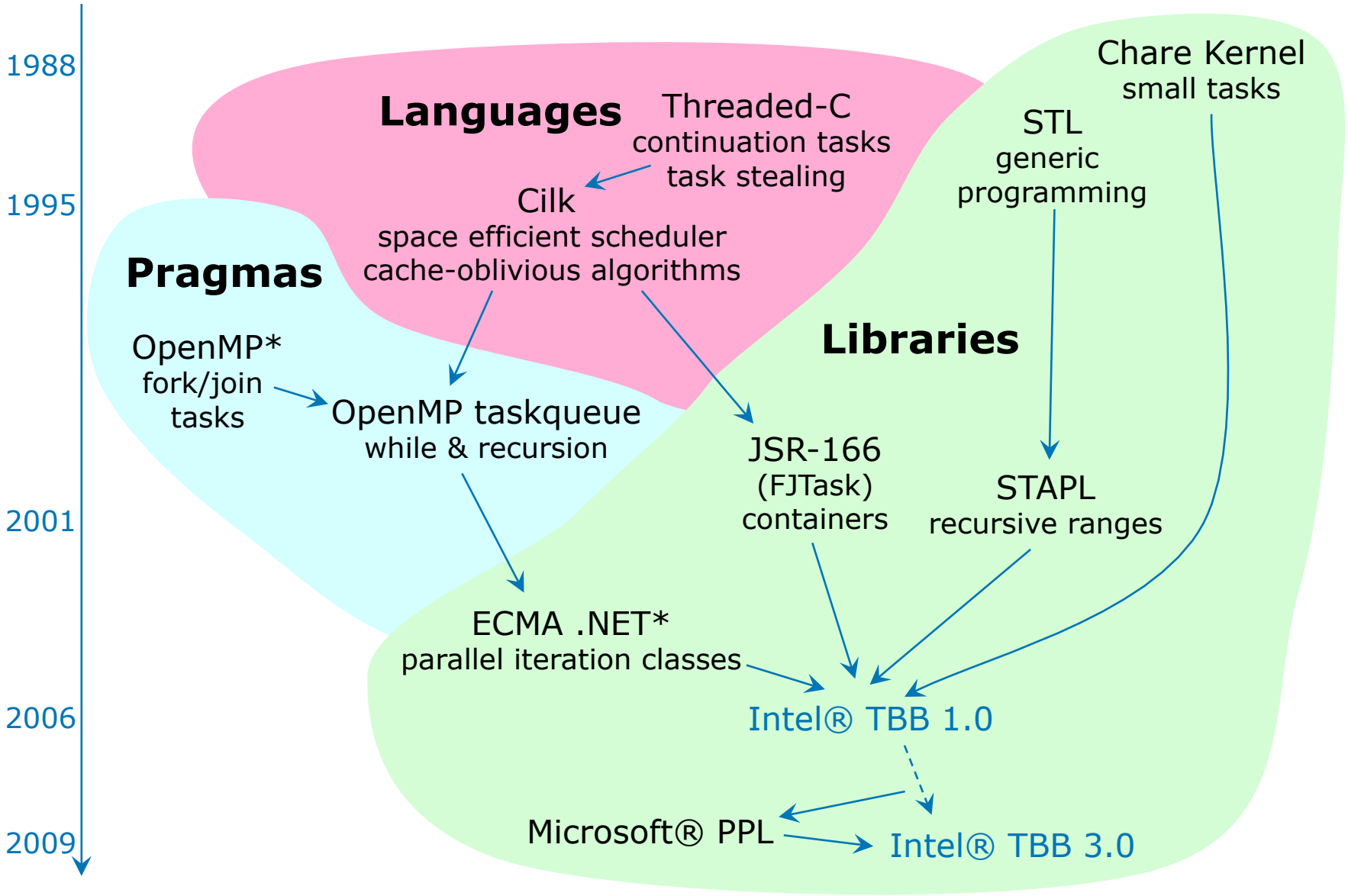


Intel® Threading Building Blocks

Michael Wrinn, Intel

ParLab Boot Camp • August 16, 2010

Family Tree



*Other names and brands may be claimed as the property of others

Key Features
Intel® Threading Building Blocks (TBB)

- 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
  - Shifts focus from workers (threads) to the work (you specify tasks patterns instead of threads
  - Hides low level details of thread management (maps your logical tasks onto physical threads)
  - Library maps your logical tasks onto physical threads, efficiently using cache and balancing load
  - Full support for nested parallelism
- It facilitates scalable performance
  - Strives for efficient use of cache, and balances load
  - Portable across Linux*, Mac OS*, Windows*, and Solaris*
  - Emphasizes scalable data parallel programming bLoop parallelism tasks are more scalable than a fixed number of separate tasks
- Can be used in concert with other packages such as native threads and OpenMP
- Open source and licensed versions available

Limitations

- 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

Intel® TBB online

www.threadingbuildingblocks.org

Downloads, active users forum, developers' blogs, documentation

Open Source License information

News and announcements

Code samples, FAQ

Address http://www.threadingbuildingblocks.org/ Go Links

Intel® Threading Building Blocks 3.0 for Open Source

Home | Downloads | Forums | Blogs | Documentation | About

Welcome Guest | Login | Register

Welcome to Threading Building Blocks.org!

Intel® Threading Building Blocks (TBB) offers a rich and complete approach to expressing parallelism in a C++ program. It is a library that helps you take advantage of multi-core processor performance without having to be a threading expert. Threading Building Blocks is not just a threads-replacement library. It represents a higher-level, task-based parallelism that abstracts platform details and threading mechanisms for scalability and performance.

May 4, 2010: Intel® Threading Building Blocks 3.0 Available Today: The new commercial aligned open source version of TBB 3.0 is available for download now. If you need the commercial version of TBB 3.0 it is available for evaluation and purchase today as well. You can learn more about the release in Terry Wilmarth's and James Reinders' blogs on the Intel Software Network.

- Read the announcement at the Intel Press Room
- Read about the history of TBB and the exciting changes available in TBB 3.0 in James Reinders' blog: TBB 3.0: New (today) Version of Intel Threading Building Blocks
- Read the summary of new features in Terry Wilmarth's blog: What's New in Intel® Threading Building Blocks 3.0
- Read about how to move from TBB 2.2 to TBB 3.0 in Arch Robison's blog: Transitioning to TBB 3.0

Intel® Threading Building Blocks *Videos* — Kevin Tureski, Product Director, Autodesk

Licensed under GPLv2 with the runtime exception.

Search Go

Page & Feed options

Bookmark This

Digg this del.icio.us

Resources

Code Samples
FAQ
Bugs
Make a Contribution
Commercial Version of TBB
Intel Community for Parallelism
Intel Open Source Software

Buy Now

Intel® Threading Building Blocks

Generic Parallel Algorithms
Efficient scalable way to exploit the power of multiple cores without having to start from scratch

Concurrent Containers
Common idioms for concurrent access - a scalable alternative to serial container with a lock around it

Thread Local Storage
Scalable implementation of thread-local data that supports infinite number of TLS

Task scheduler
The engine that empowers parallel algorithms that employs task-stealing to maximize concurrency

Synchronization Primitives
User-level and OS wrappers for mutual exclusion, ranging from atomic operations to several flavors of mutexes and condition variables

Miscellaneous
Thread-safe timers

Threads
OS API wrappers

Memory Allocation
Per-thread scalable memory manager and false-sharing free allocators

Intel® Threading Building Blocks 3.0

Generic Parallel Algorithms

parallel_for(range)
parallel_reduce
parallel_for_each(begin, end)
parallel_do
parallel_invoke
pipeline, **parallel_pipeline**
parallel_sort
parallel_scan

Concurrent Containers

concurrent_hash_map
concurrent_queue
concurrent_bounded_queue
concurrent_vector
concurrent_unordered_map

Thread Local Storage

enumerable_thread_specific
combinable

Task scheduler

task_group
task_structured_group
task_scheduler_init
task_scheduler_observer

Synchronization Primitives

atomic; mutex; recursive_mutex;
spin_mutex; spin_rw_mutex;
queuing_mutex; queuing_rw_mutex;
reader_writer_lock; critical_section;
condition_variable;
lock_guard; unique_lock;
null_mutex; null_rw_mutex;

Miscellaneous

tick_count

Threads

tbb_thread, **thread**

Memory Allocation

tbb_allocator; cache_aligned_allocator; scalable_allocator; zero_allocator

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 have been most recent in the cache
  - Oversubscription and undersubscription of core resources is prevented by task-stealing technique of TBB scheduler

Generic Programming

- Best known example is C++ STL
- Enables distribution of broadly-useful high-quality algorithms and data structures
- Write best possible algorithm with fewest constraints
  - Do not force particular data structure on user
  - Classic example: STL std::sort
- Instantiate algorithm to specific situation
  - C++ template instantiation, partial specialization, and inlining make resulting code efficient
- Standard Template Library, overall, is not *thread-safe*

Generic Programming - reminder

- 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<

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
- **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

- **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

```
#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;
}
```

Include Library Headers

Use namespace

Initialize scheduler

```
int main (){
    float A[N];
    initialize_array(A);
    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

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

Define Task

Use algorithm

Use 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

```
#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

C++0x Lambda Expression Support

parallel_for example will transform into:

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

void ChangeArrayParallel (int* a, int n )
{
    parallel_for (0, n, 1,
        [=](int i) {
            Foo (a[i]);
        });
}

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

parallel_for has an overload that takes start, stop and step argument and constructs blocked_range internally

Capture variables by value from surrounding scope to completely mimic the non-lambda implementation. Note that [&] could be used to capture variables by reference .

Using lambda expressions implement MyBody::operator() right inside the call to parallel_for().

Example using parallel_for (Game of Life) 1 of 4

Independent iterations and fixed/known bounds

```
enum State {DEAD,ALIVE} ;    // cell status
typedef State **Grid;
. . .
void NextGen(Grid oldMap, Grid newMap)
{
   int row, col;
   int ncount;
   State current;

   for (row = 1; row <= MAXROW; row++)
      for (col = 1; col <= MAXCOL; col++) {
         current = oldMap[row][col];
         ncount = NeighborCount(oldMap, row, col);
         newMap[row][col] = CellStatus(current, ncount);
      }
}
```

Example using parallel_for (Game of Life) 2 of 4

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

using namespace tbb;

typedef enum state {DEAD, ALIV                    ell status
typedef State Grid[MAXROW+2][M

. . .

int main (int argc, char *argv[])
{
    task_scheduler_init init;
. . .
}
```

Include Library Headers

Use namespace

Initialize scheduler

Example using parallel_for (Game of Life) 3 of 4

```
class CompNextGen {
  Grid oldMap;
  Grid newMap;

public:
  CompNextGen (Grid omap, Grid nmap) :
           oldMap(omap), newMap(nmap) {}

  void operator()( const blocked_range<int>& r ) const {
    for (int row = r.begin(); row < r.end(); row++)
      for (int col = 1; col <= maxcol; col++) {
        State current = oldMap[row][col];
        int ncount = NeighborCount(oldMap, row, col);
        newMap[row][col] = CellStatus(current, ncount);
      }
  }
};
```

Define Task class

operator() method is copied from original code

Example using lambdas (Game of Life) 4 of 4

- Lambdas are part C++0X proposal
- Can be used in TBB if compiler supports lambdas

```
void NextGen(Gr
{
  parallel_for (blocked_range<int>(1, maxrow+1),

      [&](const blocked_range<int>& r){
         for (int row = r.begin(); row < r.end(); row++)
           for (int col = 1; col <= MAXCOL; col++) {
             State current = oldMap[row][col];
             int ncount = NeighborCount(oldMap, row, col);
             newMap[row][col] = CellStatus(current, ncount);
           }
      }
   );
}
```

Use [&] to capture variables by reference

auto_partitioner() is default

Task Scheduler

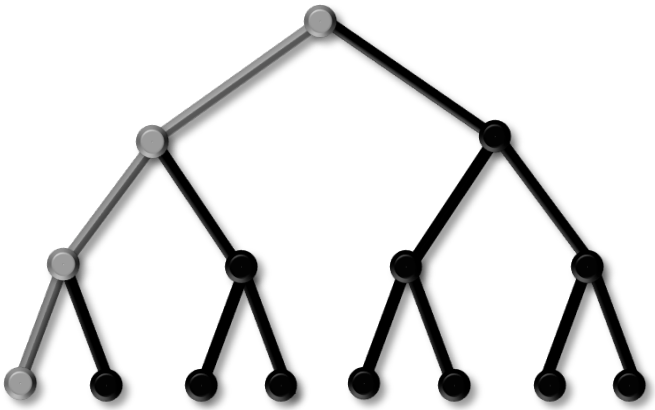
- Task scheduler is the engine driving Intel® Threading Building Blocks
  - Manages thread pool, hiding complexity of native thread management
  - Maps logical tasks to threads
- Parallel algorithms are based on task scheduler interface
- Task scheduler is designed to address common performance issues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



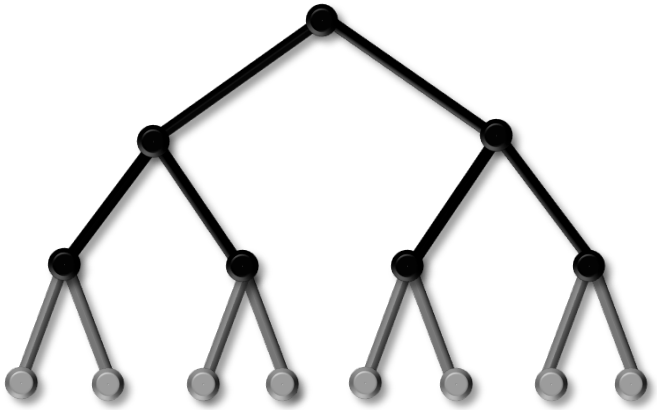
Small space

Excellent cache locality

No parallelism

Breadth First

(queue)

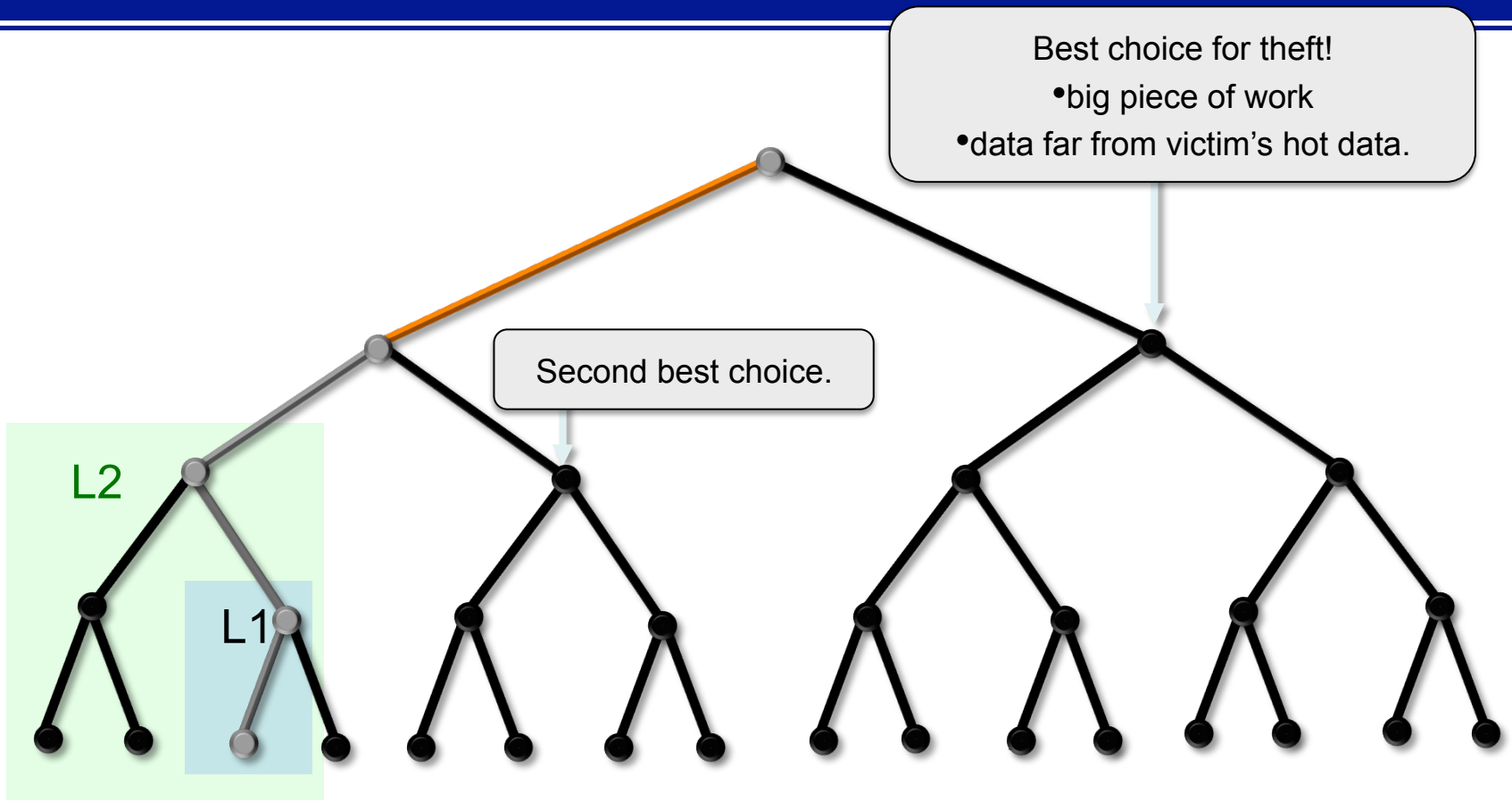


Large space

Poor cache locality

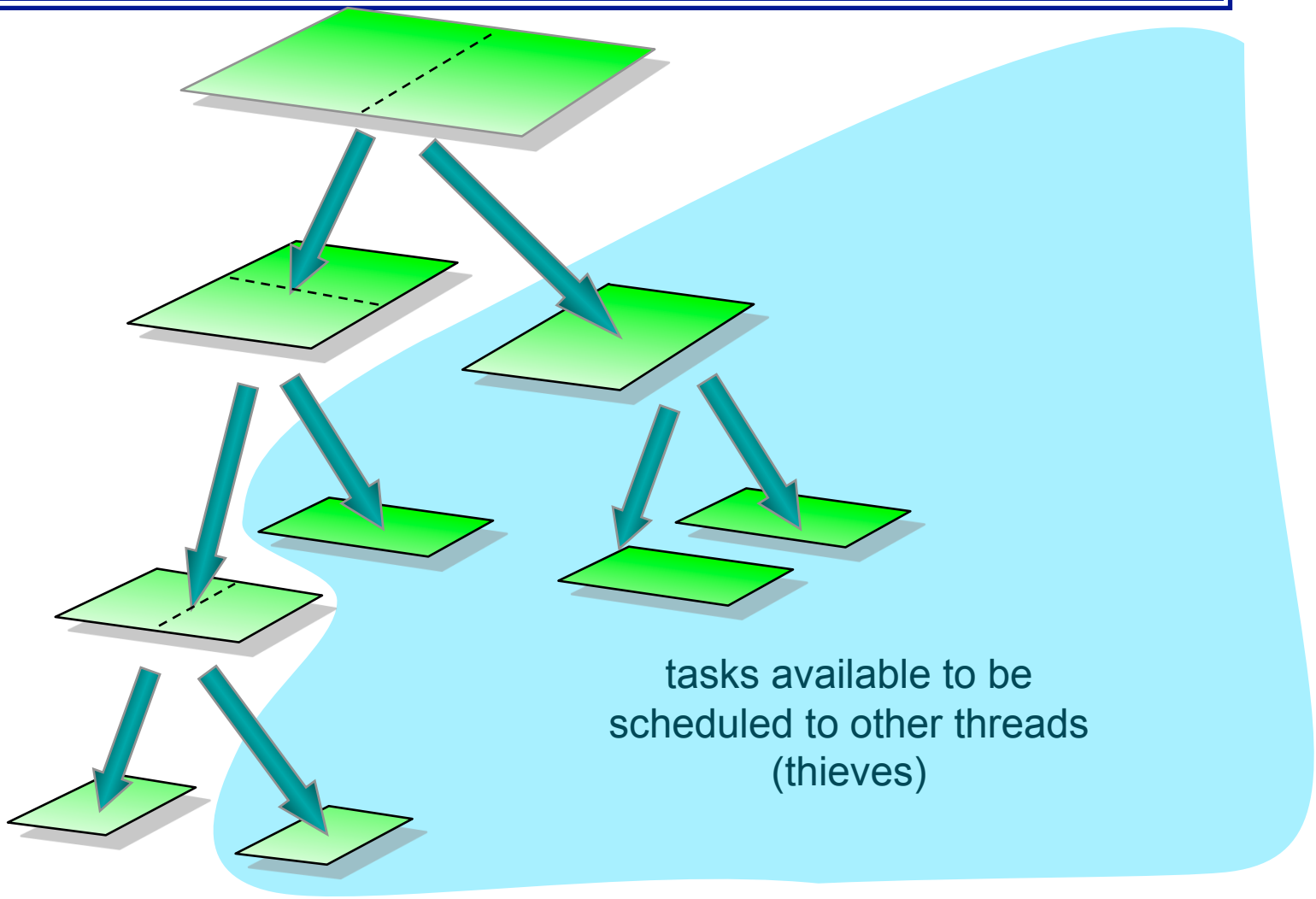
Maximum parallelism

Work Depth First; Steal Breadth First



victim thread

How splitting works on `blocked_range2d`



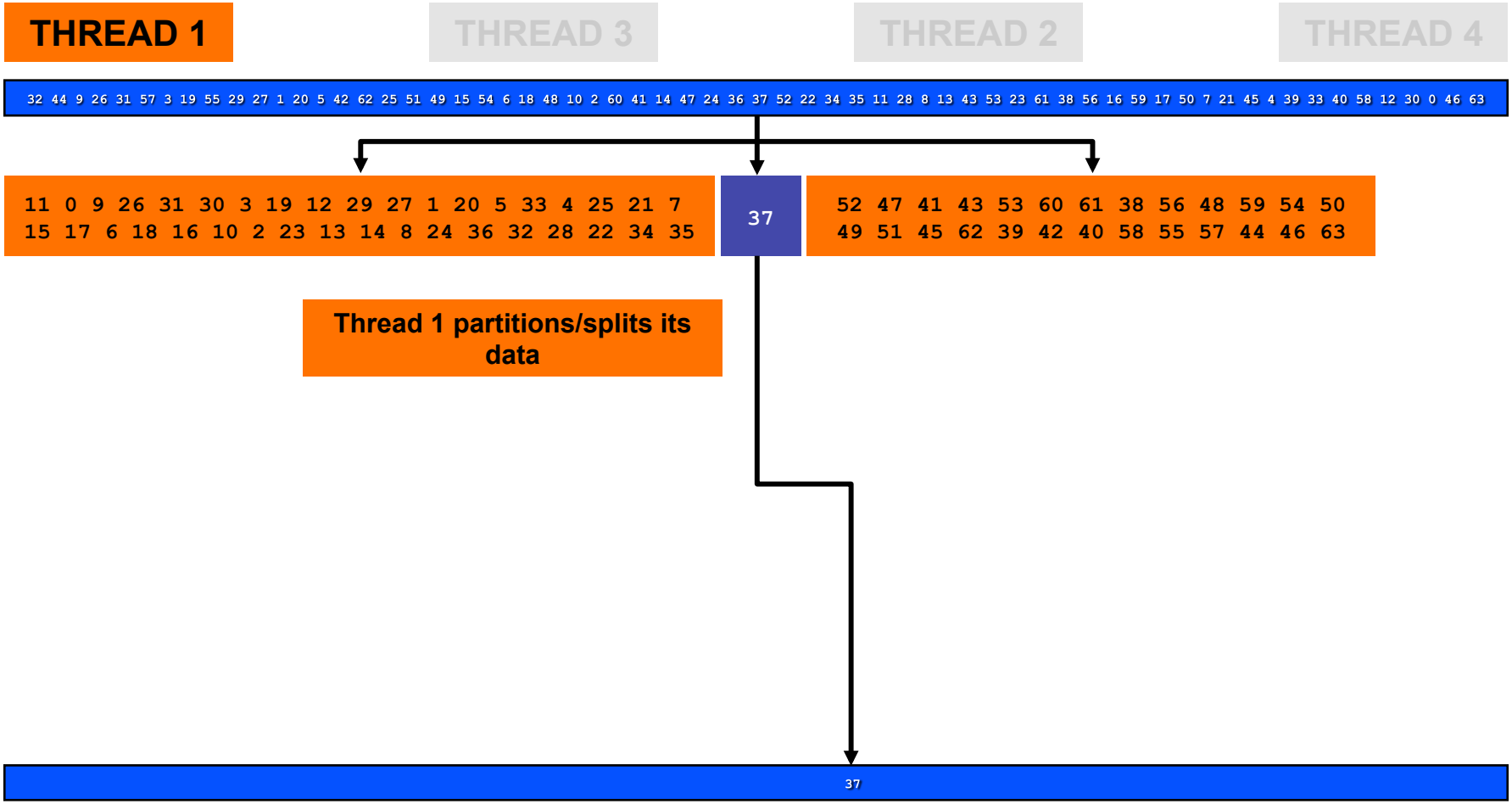
Another example: Quicksort – Step 1

THREAD 1

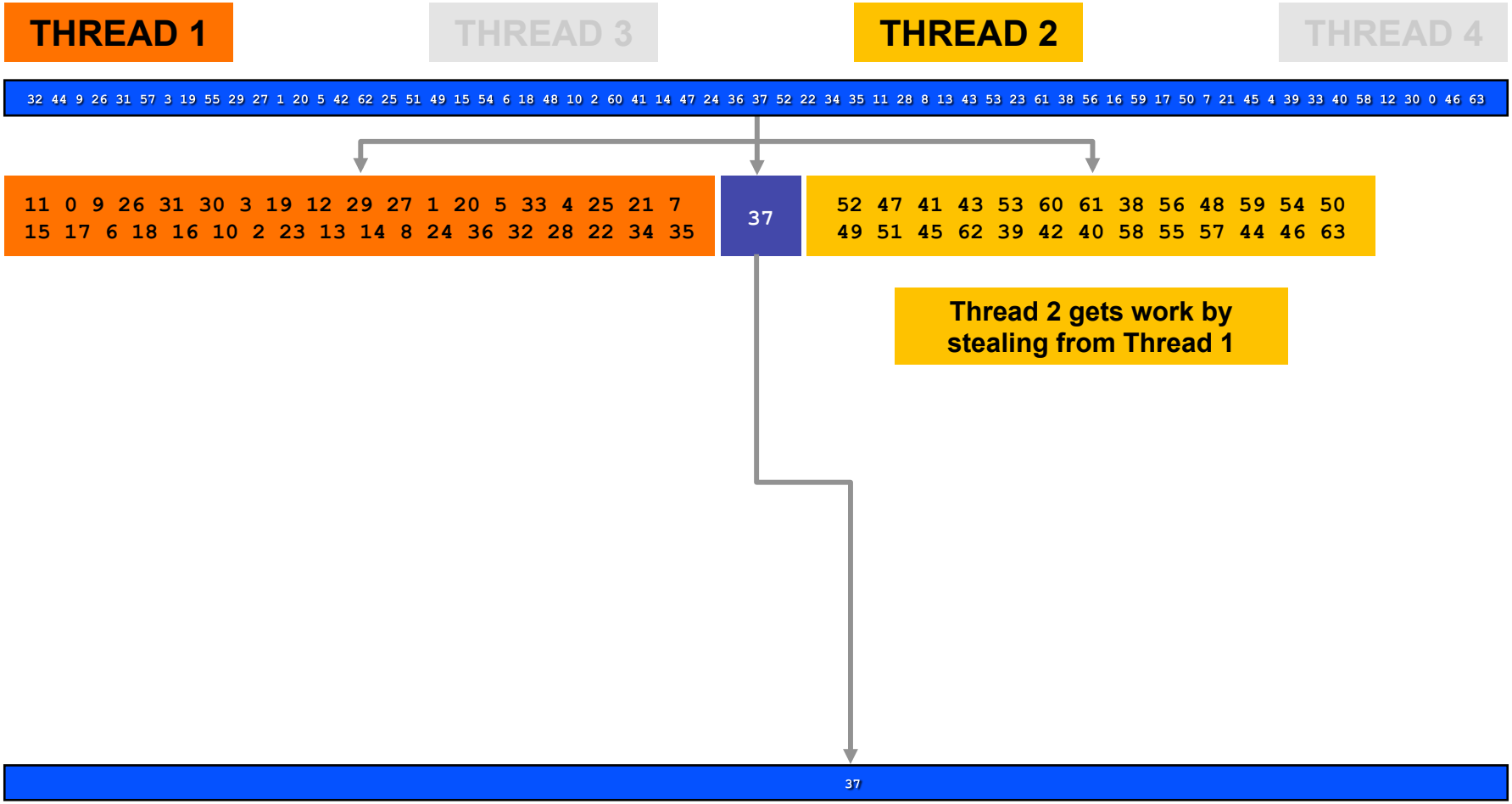
32 44 9 26 31 57 3 19 55 29 27 1 20 5 42 62 25 51 49 15 54 6 18 48 10 2 60 41 14 47 24 36 37 52 22 34 35 11 28 8 13 43 53 23 61 38 56 16 59 17 50 7 21 45 4 39 33 40 58 12 30 0 46 63

Thread 1 starts with the initial data

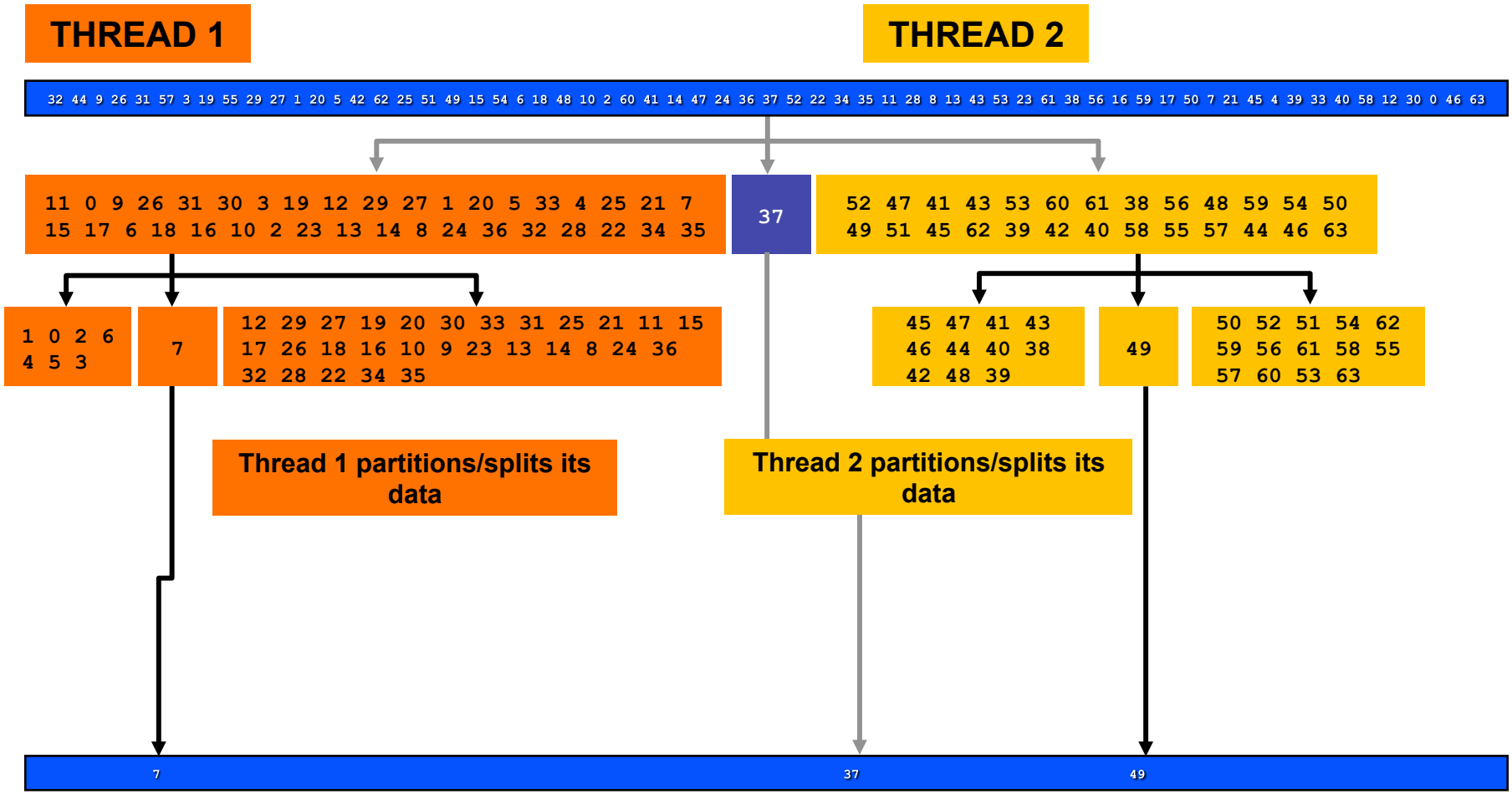
Quicksort – Step 2



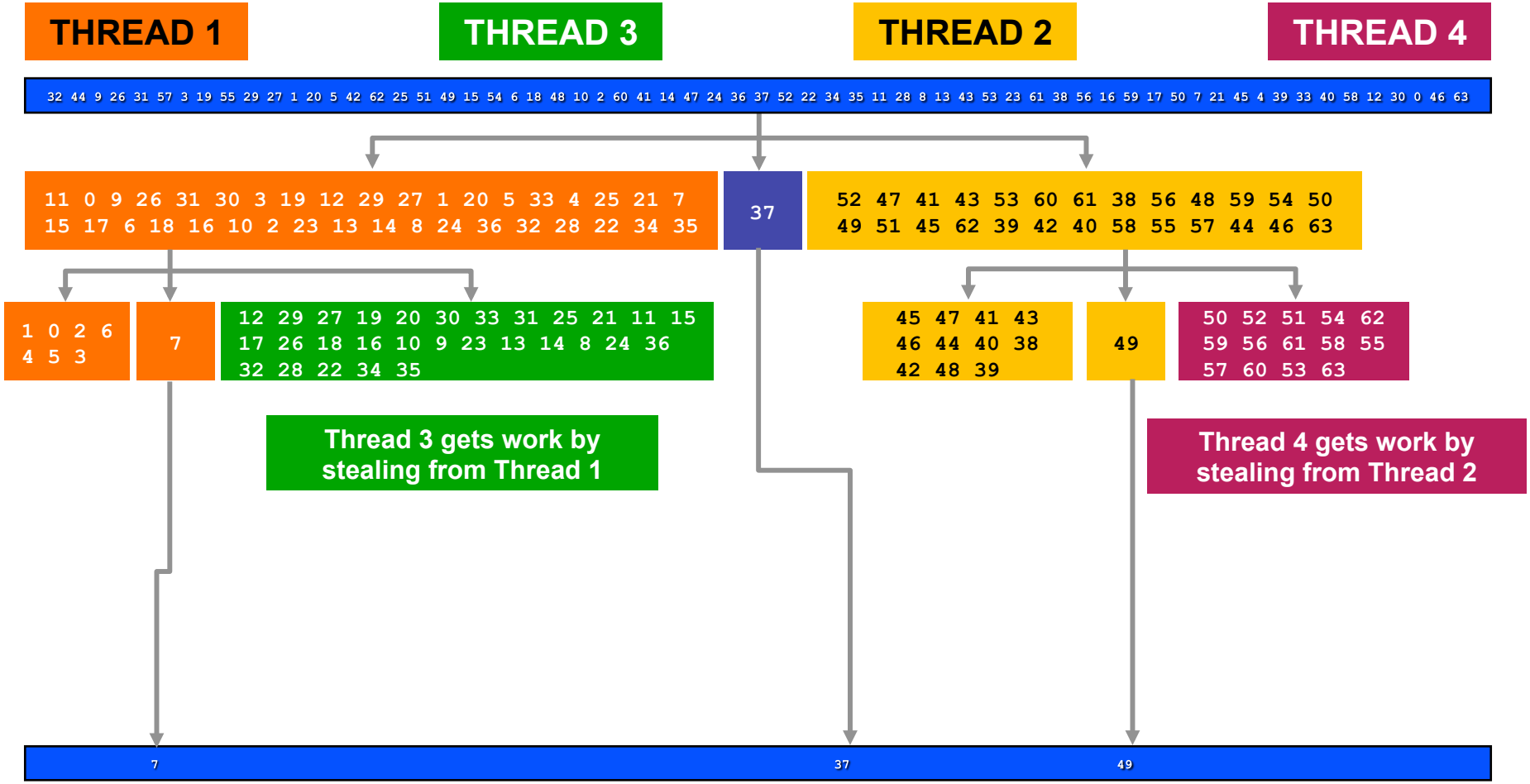
Quicksort – Step 2



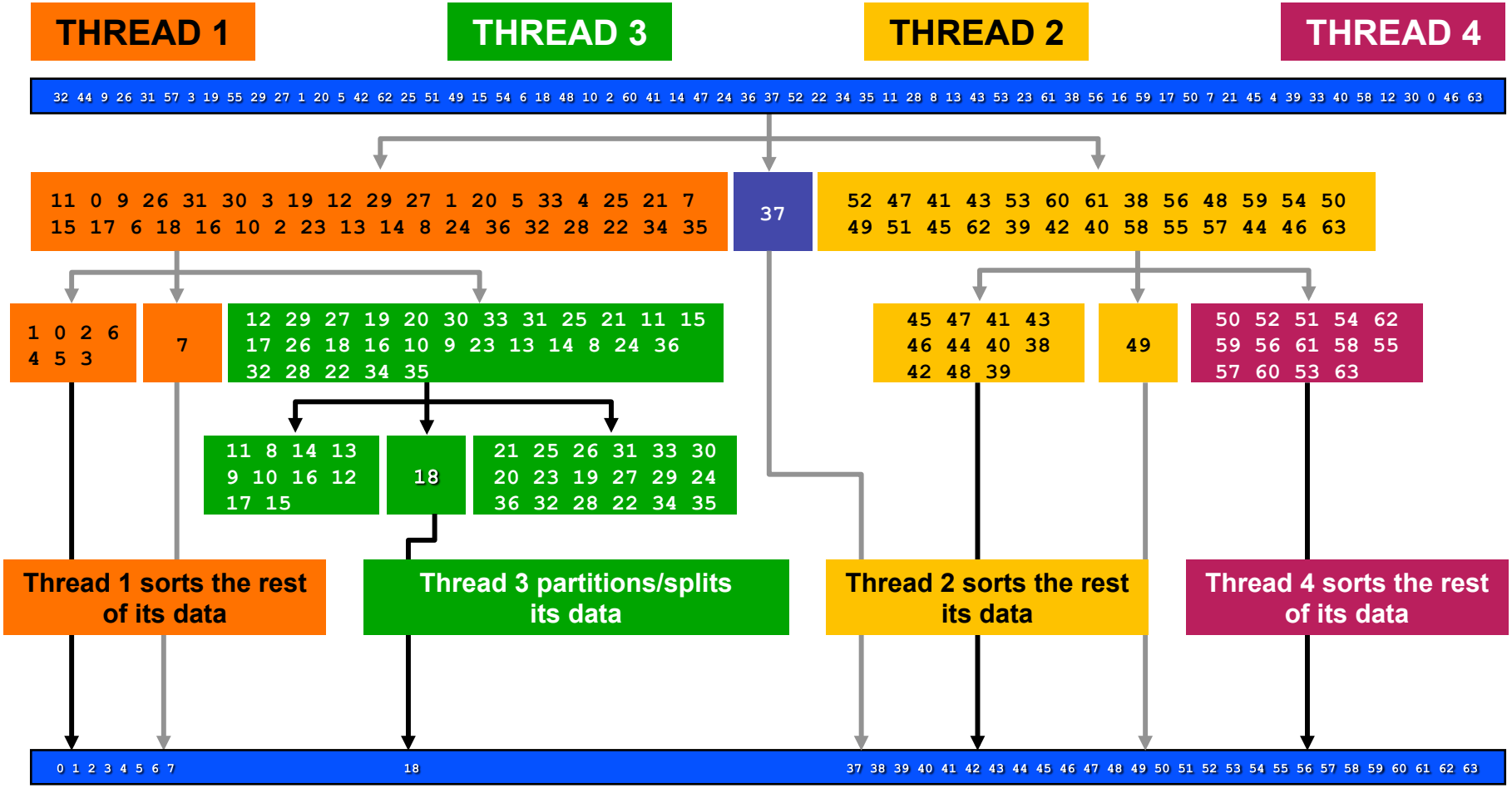
Quicksort – Step 3



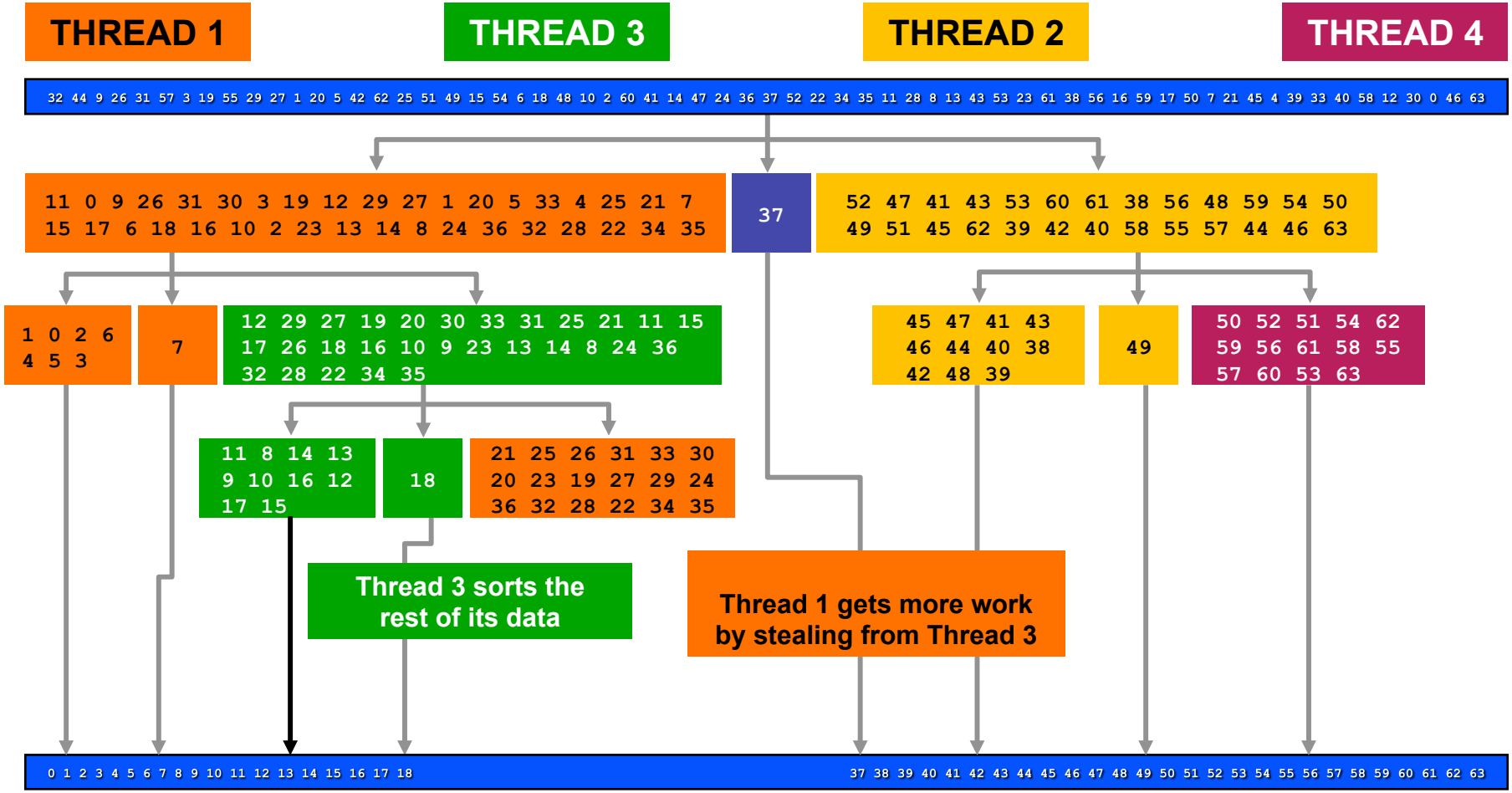
Quicksort – Step 3



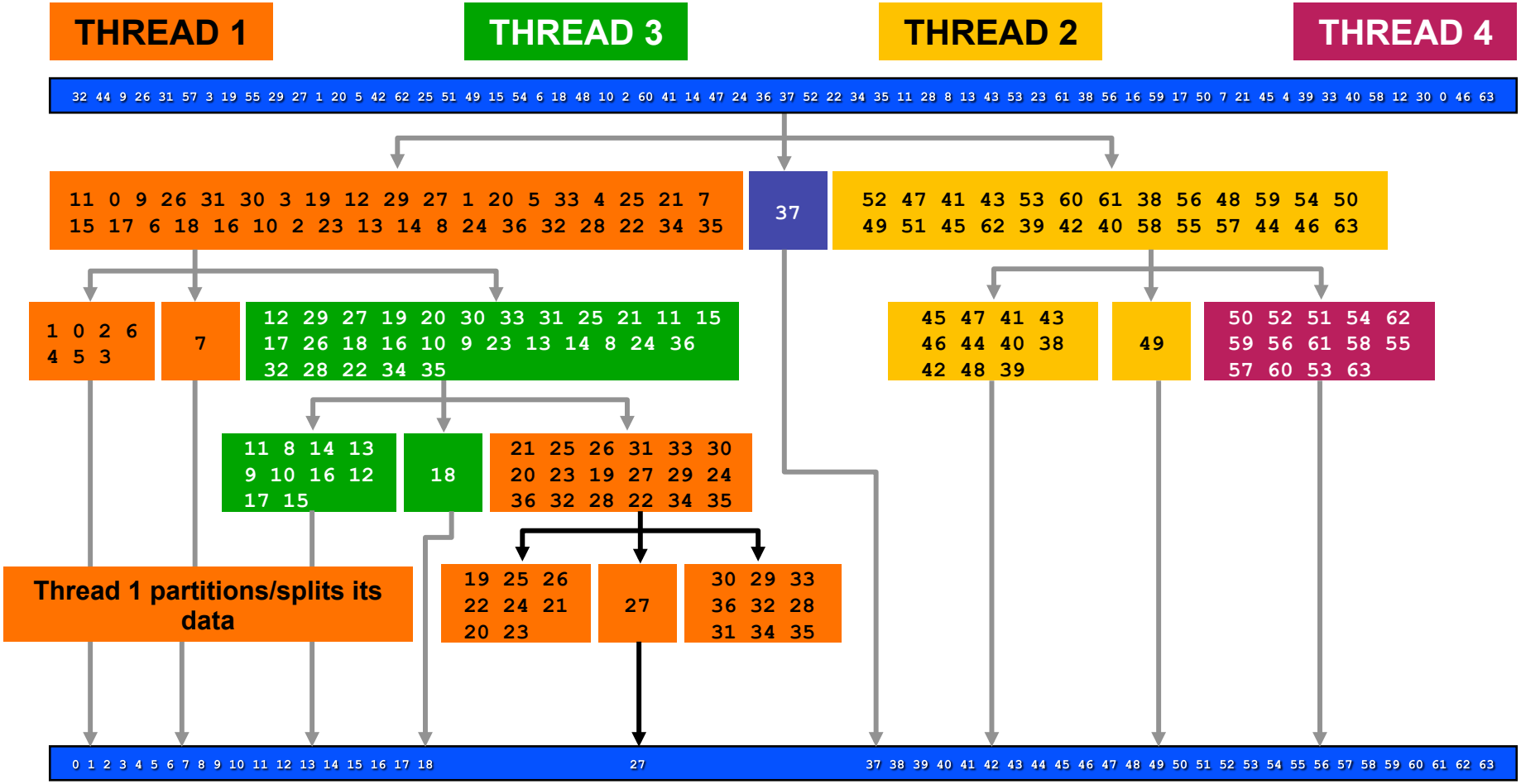
Quicksort – Step 4



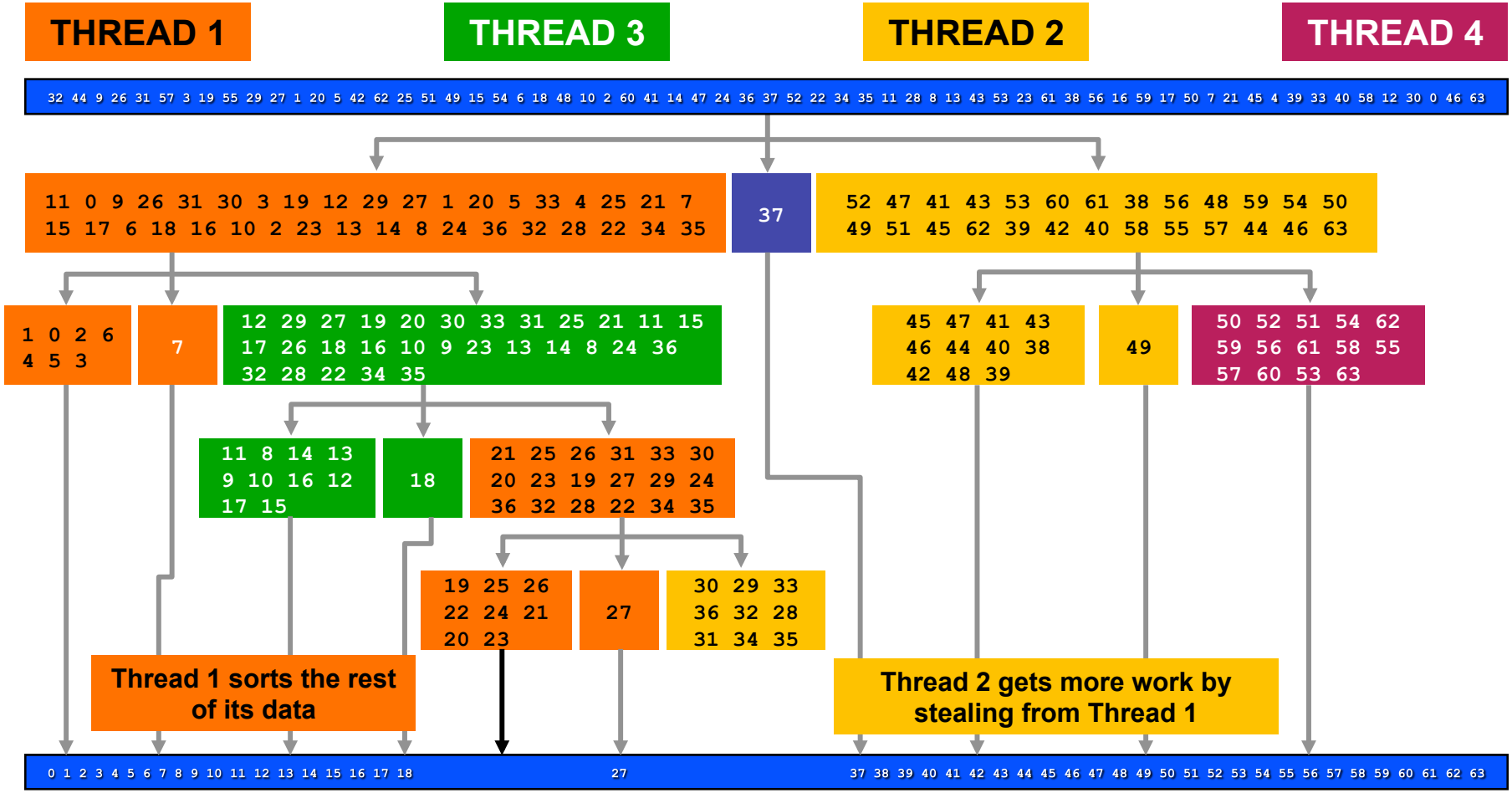
Quicksort – Step 5



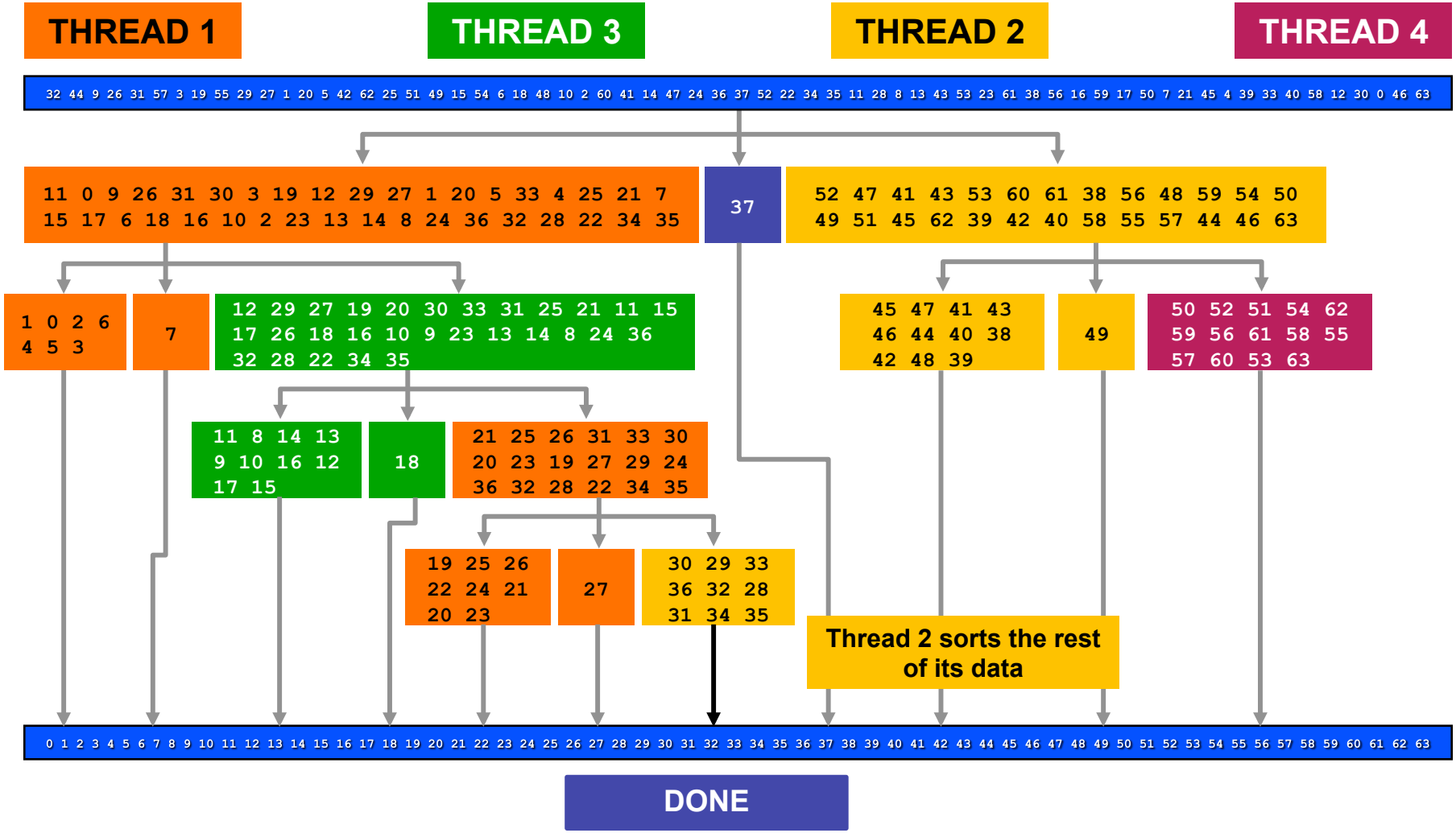
Quicksort – Step 6



Quicksort – Step 6



Quicksort – Step 7



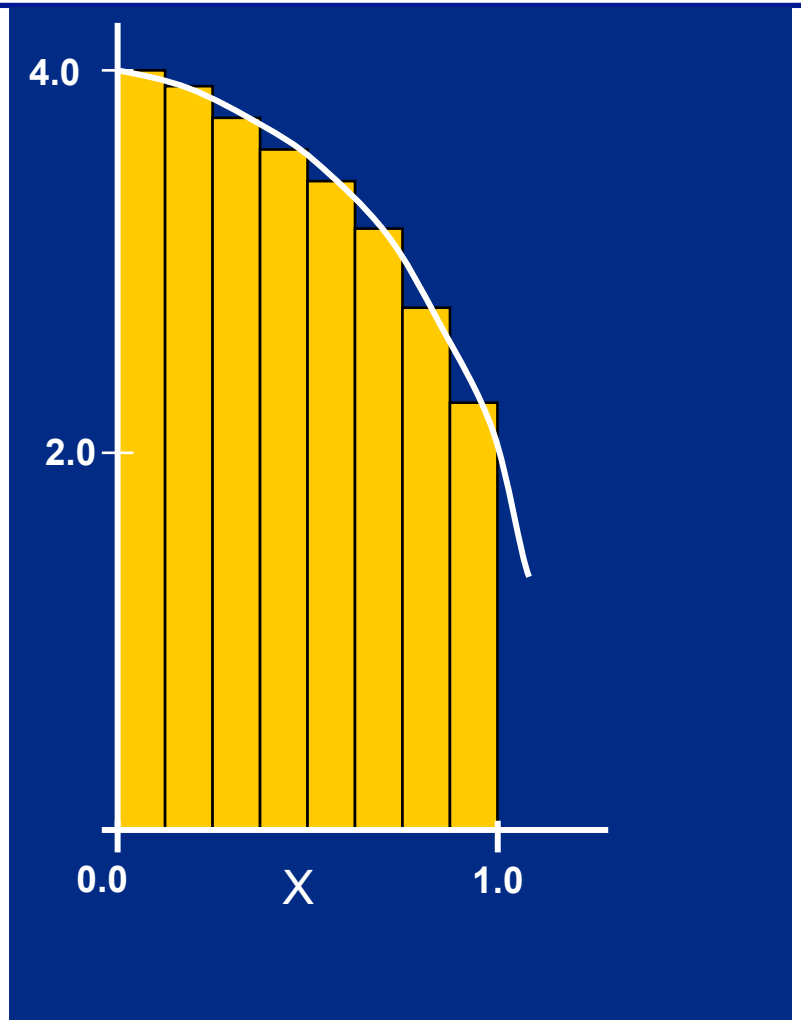
The parallel_reduce Template

```
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)




```
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

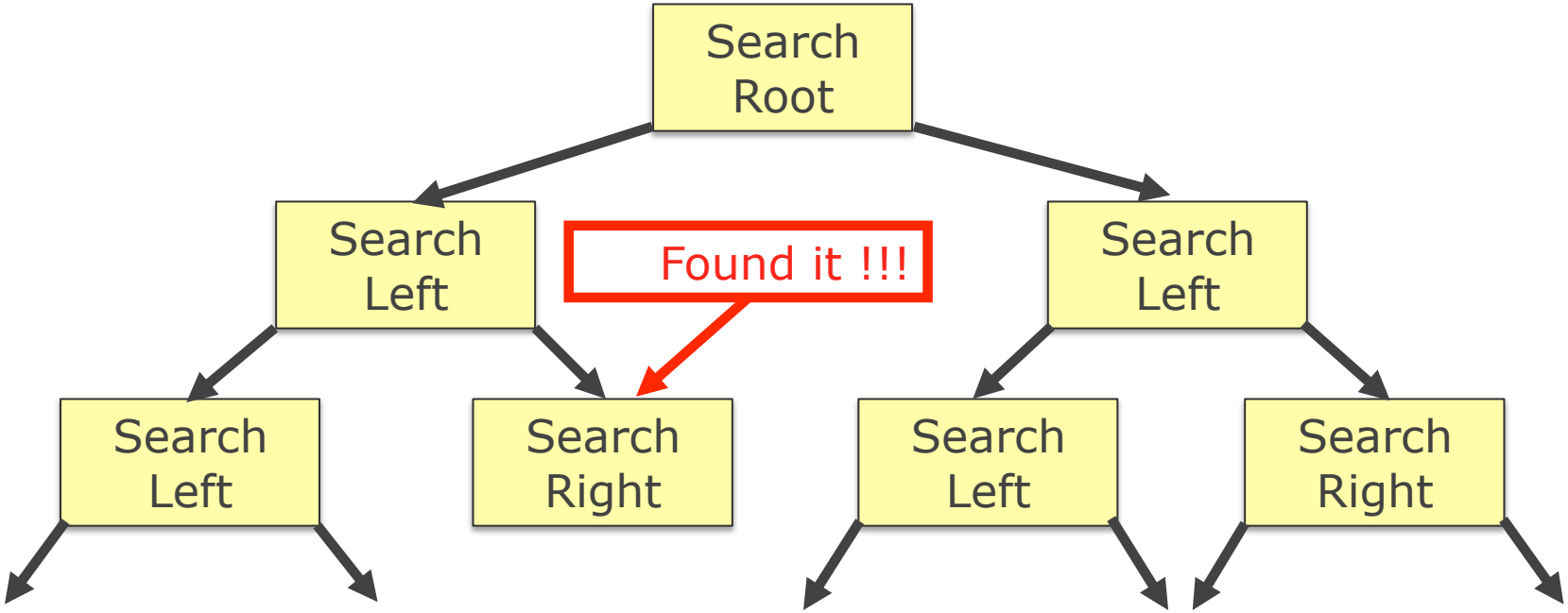
join

lambda version of pi / parallel_reduce
-- thx to Alexy Kukanov, from TBB forum

```
int main(int argc, char* argv[])
{
  double pi;
  double width = 1./(double)num_steps;

  pi = parallel_reduce(
         blocked_range<size_t>(0, num_steps),
         double(0),
         [&]( blocked_range<size_t>& r, double current_sum ) -> double {
                 for (size_t i = r.begin(); i < r.end(); ++i) {
                     double x = (i+0.5)*step;
                     current_sum += 4.0/(1.0 + x*x);
                 }
                 return current_sum;
        },
        [](double s1, double s2 ) {
              return s1+s2;       // "joins" two accumulated values
           }
        );

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

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

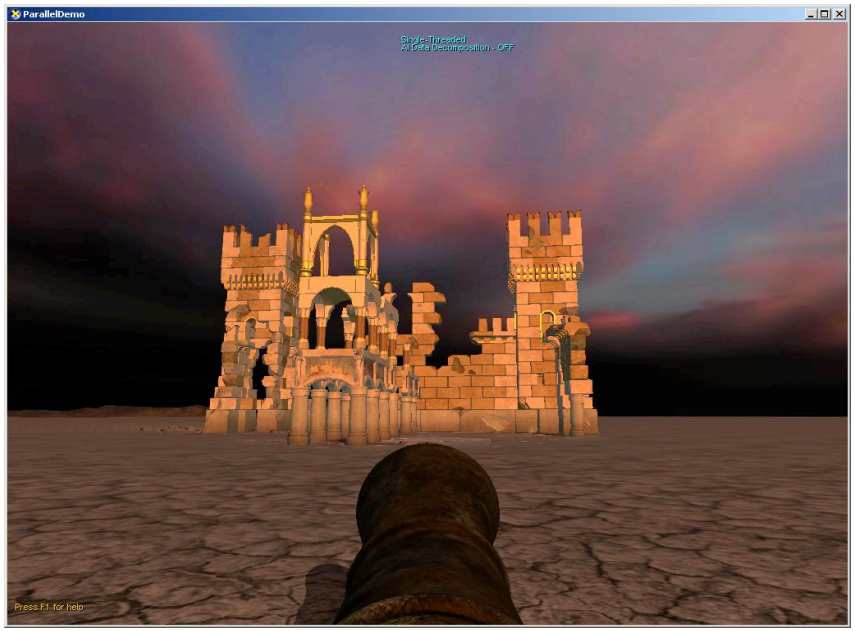
```
int main() {

   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

Improving data locality with task-to-thread affinity

(using Destroy the Castle demo as an example)



- "Destroy the Castle" computes Artificial Intelligence, Particles and Physics for moving objects for each frame by executing corresponding TBB tasks.
- Localizing computations of each type on one particular thread, but at the same time being able to balance the load is very important

http://www3.intel.com/cd/ids/developer/asmo-na/eng/337398.htm

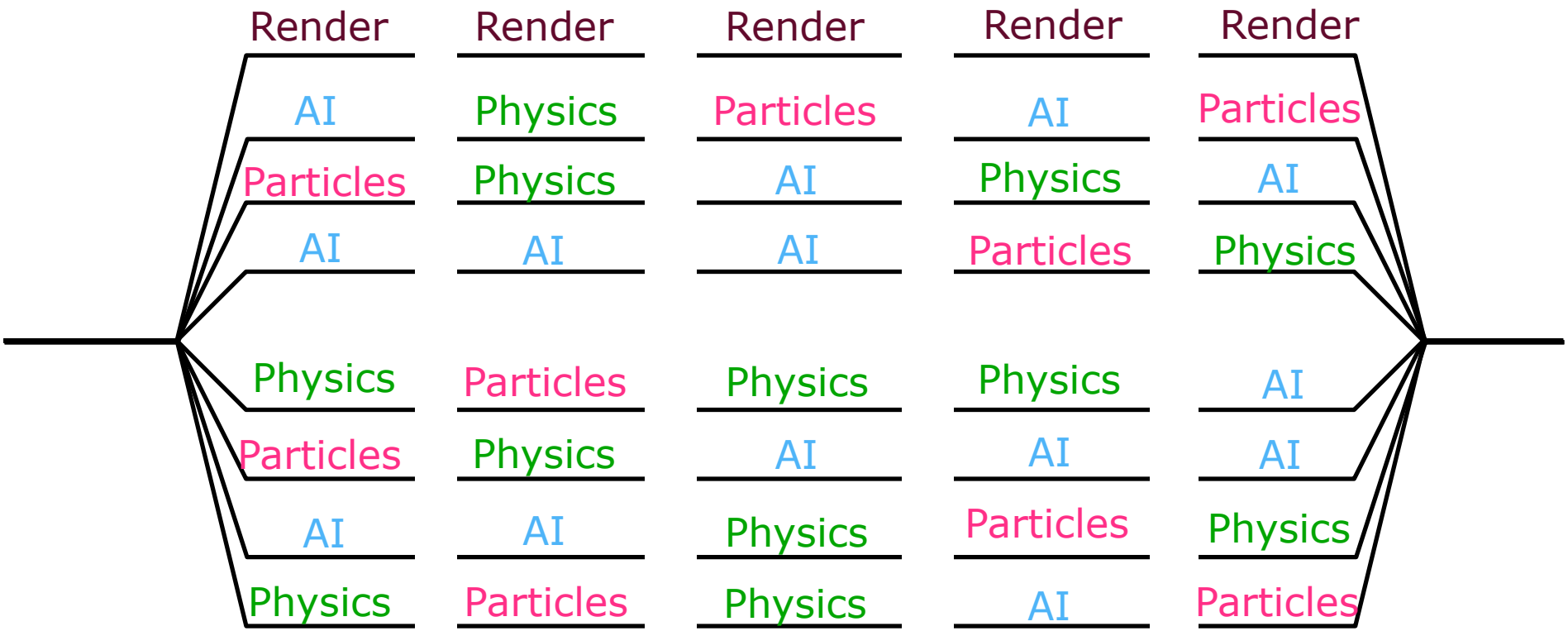
Traditional approach to game parallelization vs. the Intel® Threading Building Blocks approach

- Traditionally the number of threads is hardwired in the code:
  - + Better data locality
  - - Virtually no scalability
- Intel TBB approach is to subdivide into smaller tasks and run in parallel on an optimal number of threads
  - + "Free" scalability
  - - Poor data locality

Affinity partitioner maps tasks to threads and stores "affinity" information in between iterations. This is why it has to be declared outside the nesting for-loop

```
affinity_partitioner partitioner;

for ( i = 0; i < N_iter; ++i ) {
    parallel_for( blocked_range<int>(0, num_bodies),
      AIBody(...), partitioner );
}
```

Effect of task-to-thread affinity



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

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.

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 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

- 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

spin_mutex Example

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

```
#include "tbb/spin_mutex.h"
Node* FreeList;
typedef spin_mutex FreeListMutexType;
FreelistMutexType FreelistMutex;

Node* AllocateNode (){
  Node* n;
  {
    FreelistMutexType::scoped_lock mylock(FreeListMutex);
    n = FreeList;
    if ( n ) FreeList = n->next;
  }
  if ( !n ) n = new Node();
  return n;
}

void FreeNode( Node* n ) {
  FreelistMutexType::scoped_lock mylock(FreeListMutex);
  n->next = FreeList;
  FreeList = n;
}
```

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