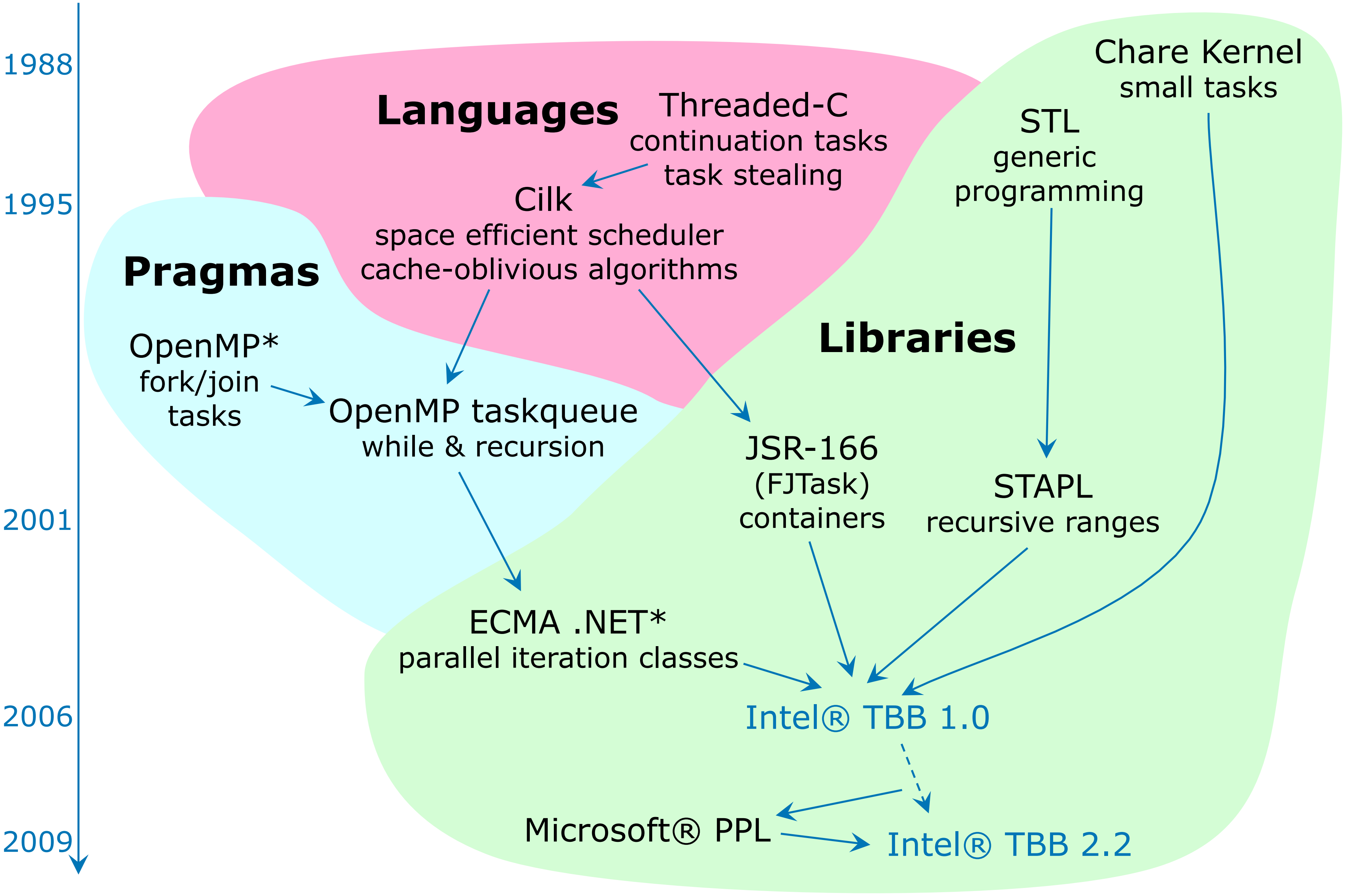
Intel® Threading Building Blocks

Michael Wrinn, Intel

ParLab Boot Camp • August 19, 2009

Family Tree



1988

1995

2001

2006

2009

Languages

Threaded-C
continuation tasks
task stealing

Cilk
space efficient scheduler
cache-oblivious algorithms

Pragmas

OpenMP*
fork/join
tasks

OpenMP taskqueue
while & recursion

ECMA .NET*
parallel iteration classes

Libraries

Chare Kernel
small tasks

STL
generic
programming

JSR-166
(FJTask)
containers

STAPL
recursive ranges

Intel® TBB 1.0

Microsoft® PPL

Intel® TBB 2.2

*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

Check Intel® TBB online

Welcome Guest | Login | Register

Welcome to Threading Building Blocks.org!

Intel® Threading … elism in a
C++ program. It i… without
having to be a thre… ry. It
represents a highe…
mechanisms for sc…

Intel® Threading Building Blocks 2.2 Commercial Version Available Now: The commercial version is available to evaluate and the new commercially aligned release is available for download now. Learn more about the great additions to TBB in 2.2 in Terry Wilmarth's and James Reinders' blogs on the Intel Software Network.

- Read Intel's Announcement
- Read Terry Wilmarth's blog What's New in Intel® TBB 2.2?
- Read James Reinders' blogs about 2.2:

Blocks

Games,
.2 has
has joined

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

Active user forums, FAQs, technical blogs and TBB Developers Wiki

Several very important contributions were made by the OS community allowing TBB 2.1 to build and work on: XBox* 360, Sun Solaris*, AIX*

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 2.2 Components

Generic Parallel Algorithms

parallel_for, parallel_for_each
parallel_reduce
parallel_scan
parallel_do
pipeline
parallel_sort
parallel_invoke

Task scheduler

task_group
task
task_scheduler_init
task_scheduler_observer

Synchronization Primitives

atomic, mutex, recursive_mutex
spin_mutex, spin_rw_mutex
queuing_mutex, queuing_rw_mutex
null_mutex, null_rw_mutex

Threads

tbb_thread

Concurrent Containers

concurrent_hash_map
concurrent_queue
concurrent_bounded_queue
concurrent_vector

Thread Local Storage

combinable
enumerable_thread_specific

Memory Allocation

tbb_allocator
zero_allocator
cache_aligned_allocator
scalable_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 - 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<

TBB Parallel Algorithms

- Task scheduler powers high level parallel patterns that are pre-packaged, tested, and tuned for scalability
  - parallel_for: 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_do: load-balanced parallel execution of independent loop iterations with unknown or dynamically changing bounds (e.g. applying function to the element of linked list)
  - parallel_scan: template function that computes parallel prefix
  - pipeline: data-flow pipeline pattern
  - parallel_sort: parallel sort
  - parallel_invoke: evaluates up to 10 functions, possibly in parallel and waits for all of them to finish.

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

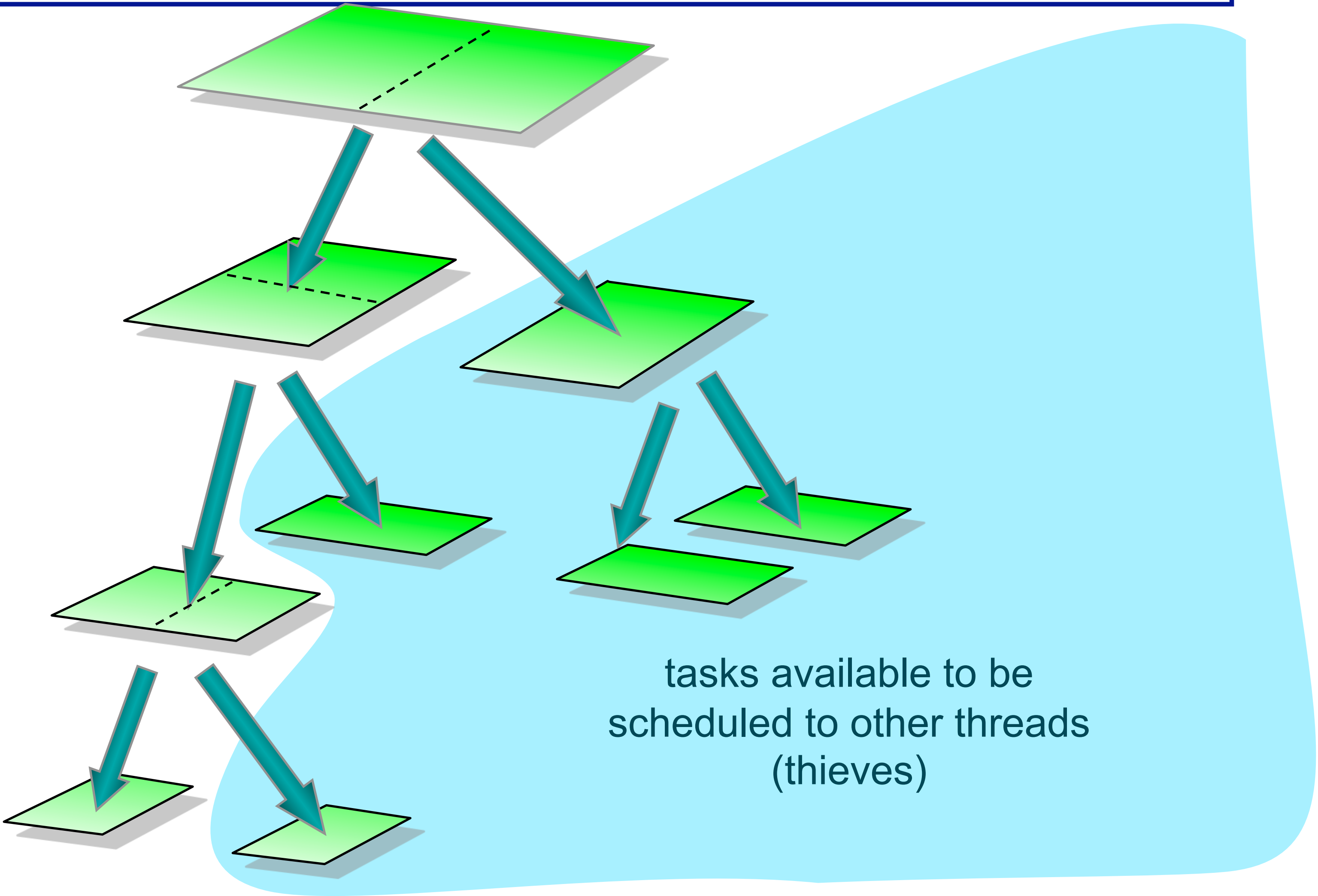
Range is Generic

- 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

How splitting works on `blocked_range2d`



tasks available to be scheduled to other threads (thieves)

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

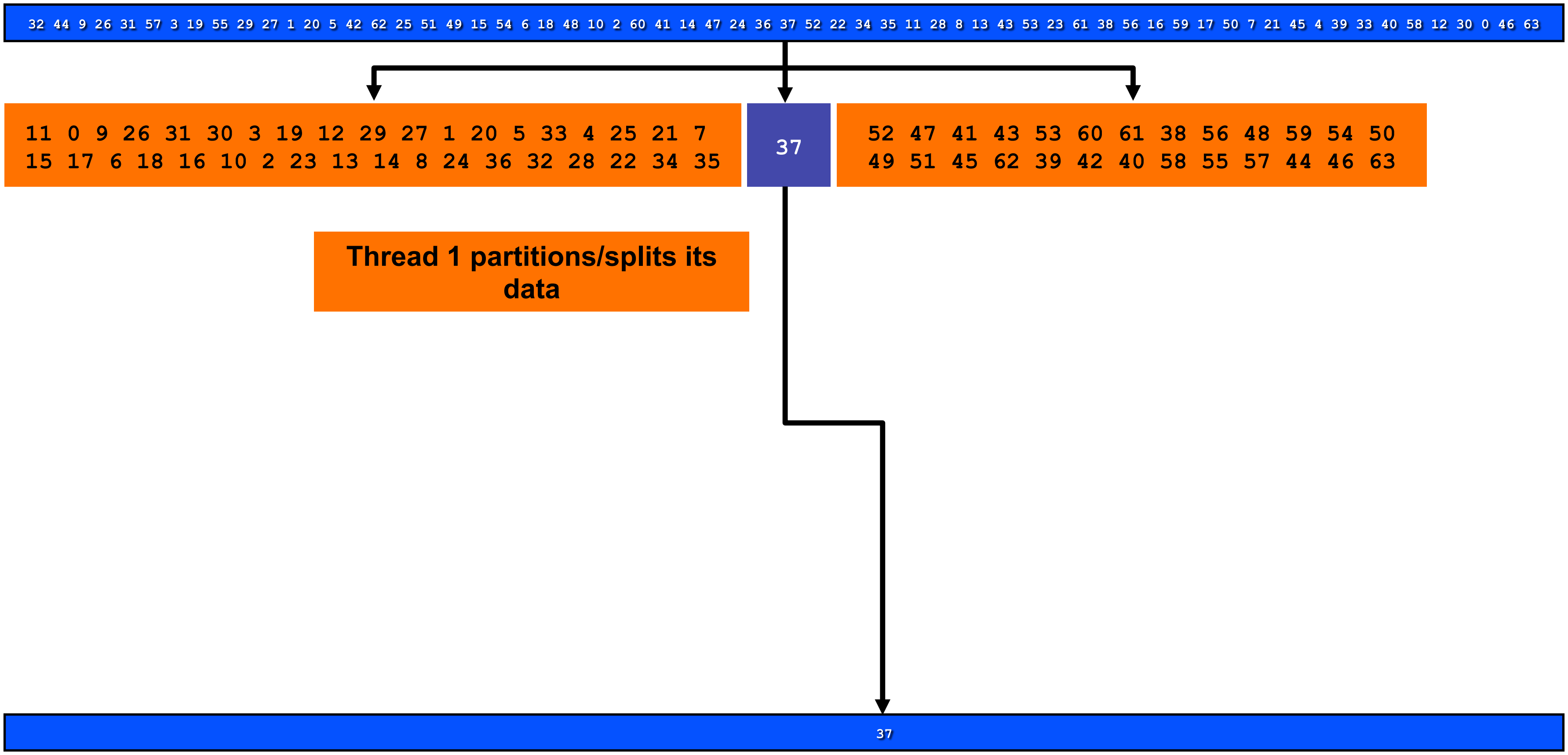
Quicksort – Step 2

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

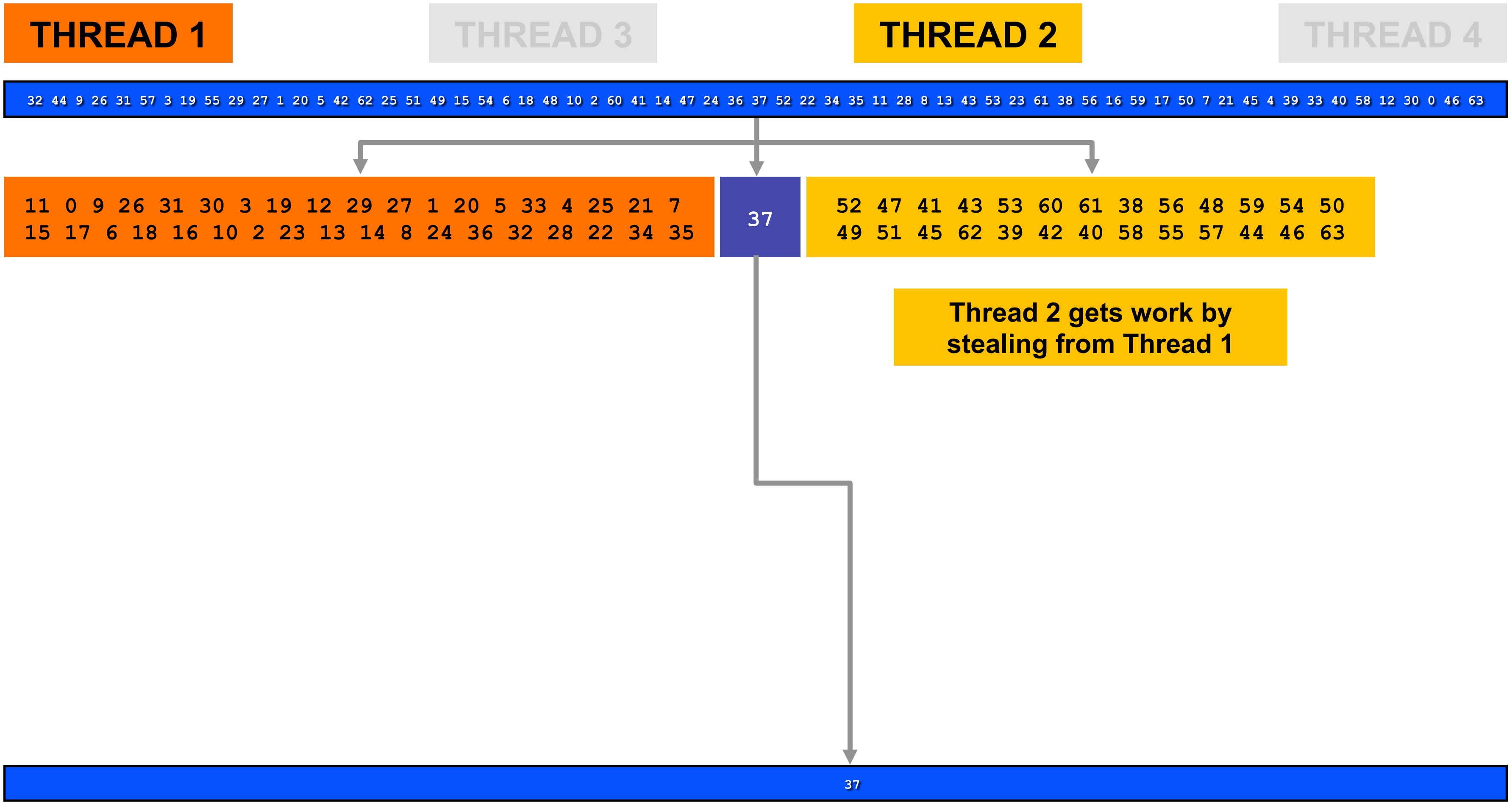
37

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

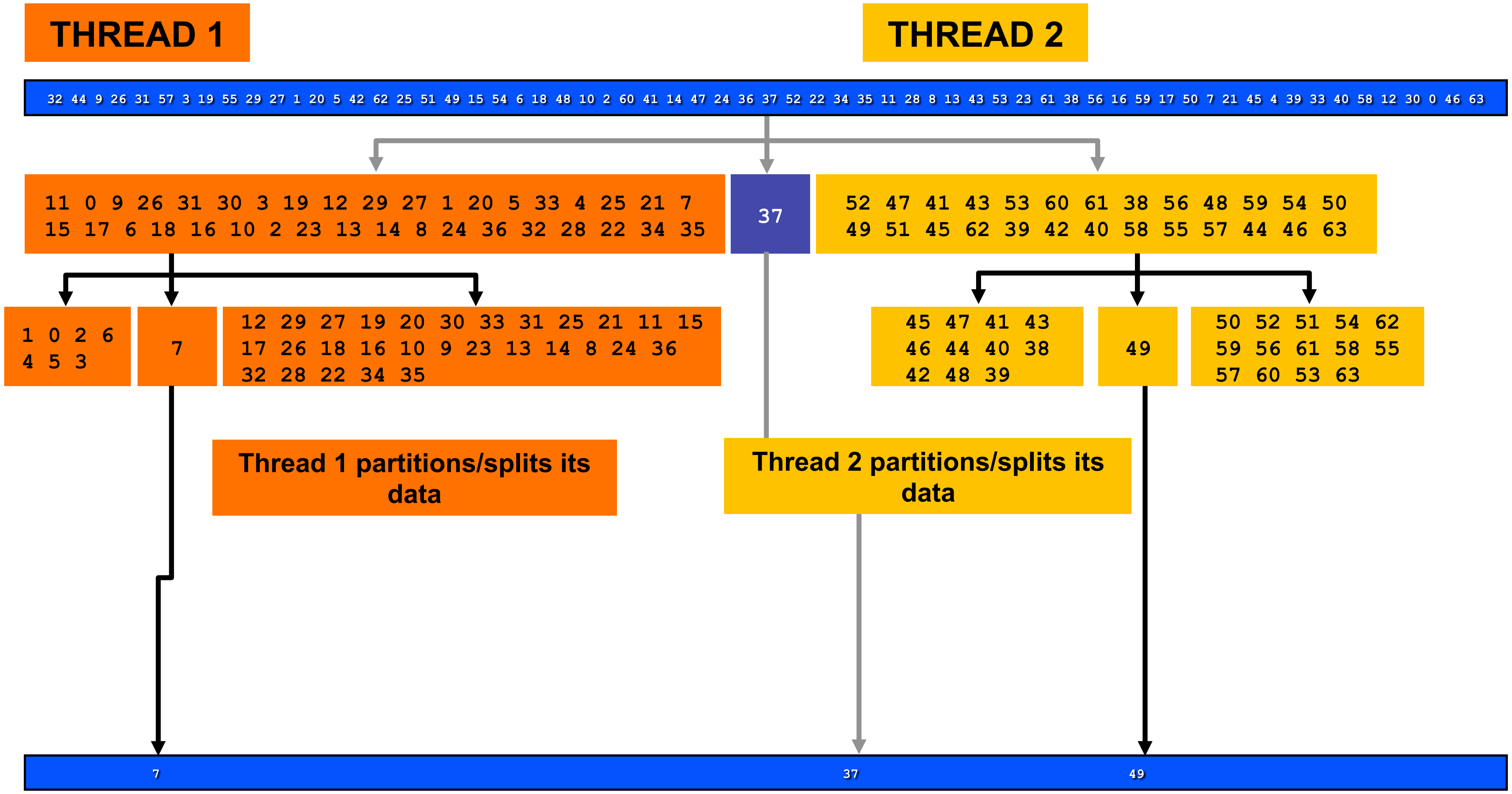
37

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



THREAD 1

THREAD 2

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

1 0 2 6 4 5 3

7

12 29 27 19 20 30 33 31 25 21 11 15 17 26 18 16 10 9 23 13 14 8 24 36 32 28 22 34 35

45 47 41 43 46 44 40 38 42 48 39

49

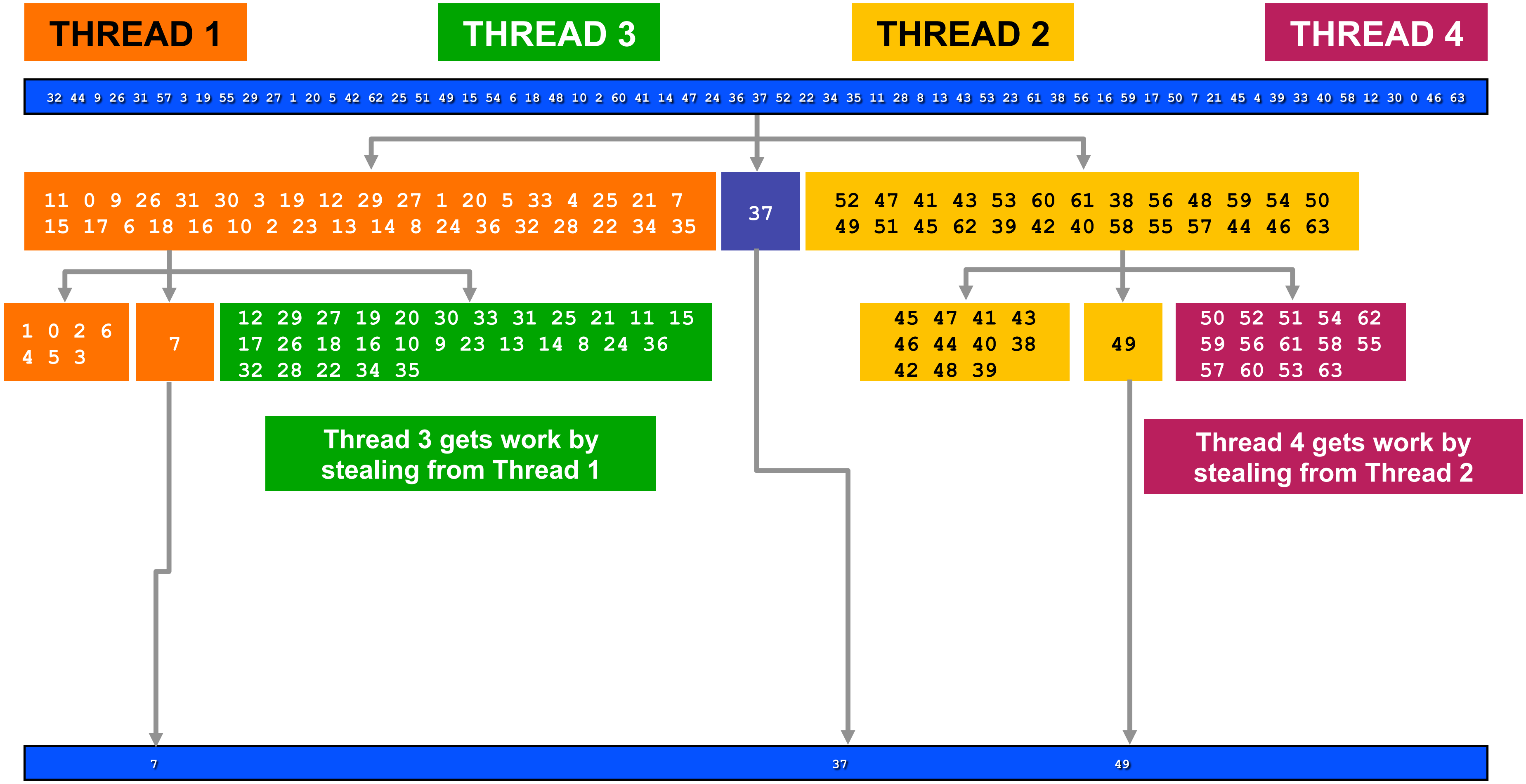
50 52 51 54 62 59 56 61 58 55 57 60 53 63

Thread 1 partitions/splits its data

Thread 2 partitions/splits its data

7 37 49

Quicksort – Step 3



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

1 0 2 6
4 5 3

7

12 29 27 19 20 30 33 31 25 21 11 15
17 26 18 16 10 9 23 13 14 8 24 36
32 28 22 34 35

45 47 41 43
46 44 40 38
42 48 39

49

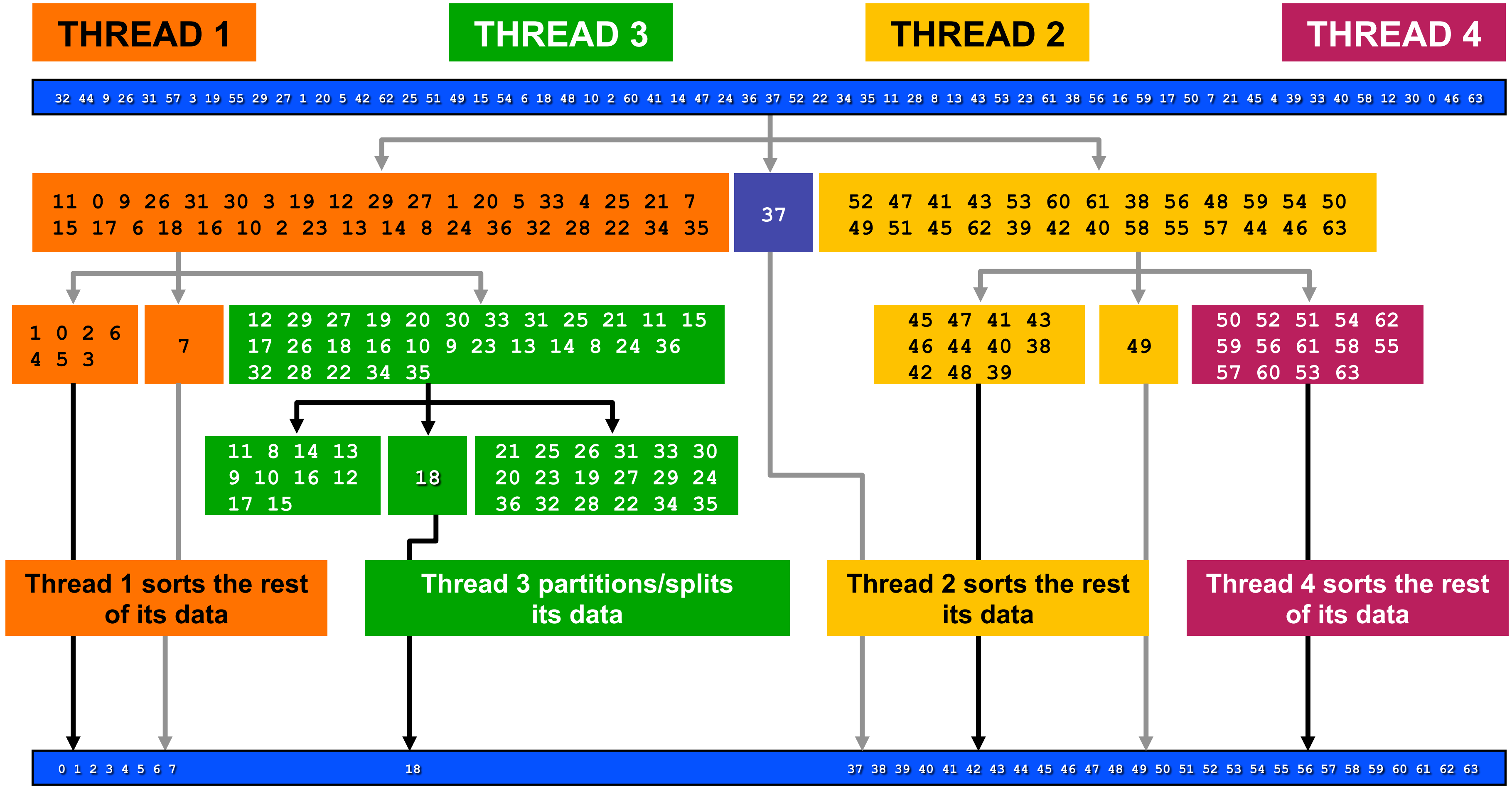
50 52 51 54 62
59 56 61 58 55
57 60 53 63

Thread 3 gets work by stealing from Thread 1

Thread 4 gets work by stealing from Thread 2

7 37 49

Quicksort – Step 4



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

1 0 2 6
4 5 3

7

12 29 27 19 20 30 33 31 25 21 11 15
17 26 18 16 10 9 23 13 14 8 24 36
32 28 22 34 35

45 47 41 43
46 44 40 38
42 48 39

49

50 52 51 54 62
59 56 61 58 55
57 60 53 63

11 8 14 13
9 10 16 12
17 15

18

21 25 26 31 33 30
20 23 19 27 29 24
36 32 28 22 34 35

Thread 1 sorts the rest of its data

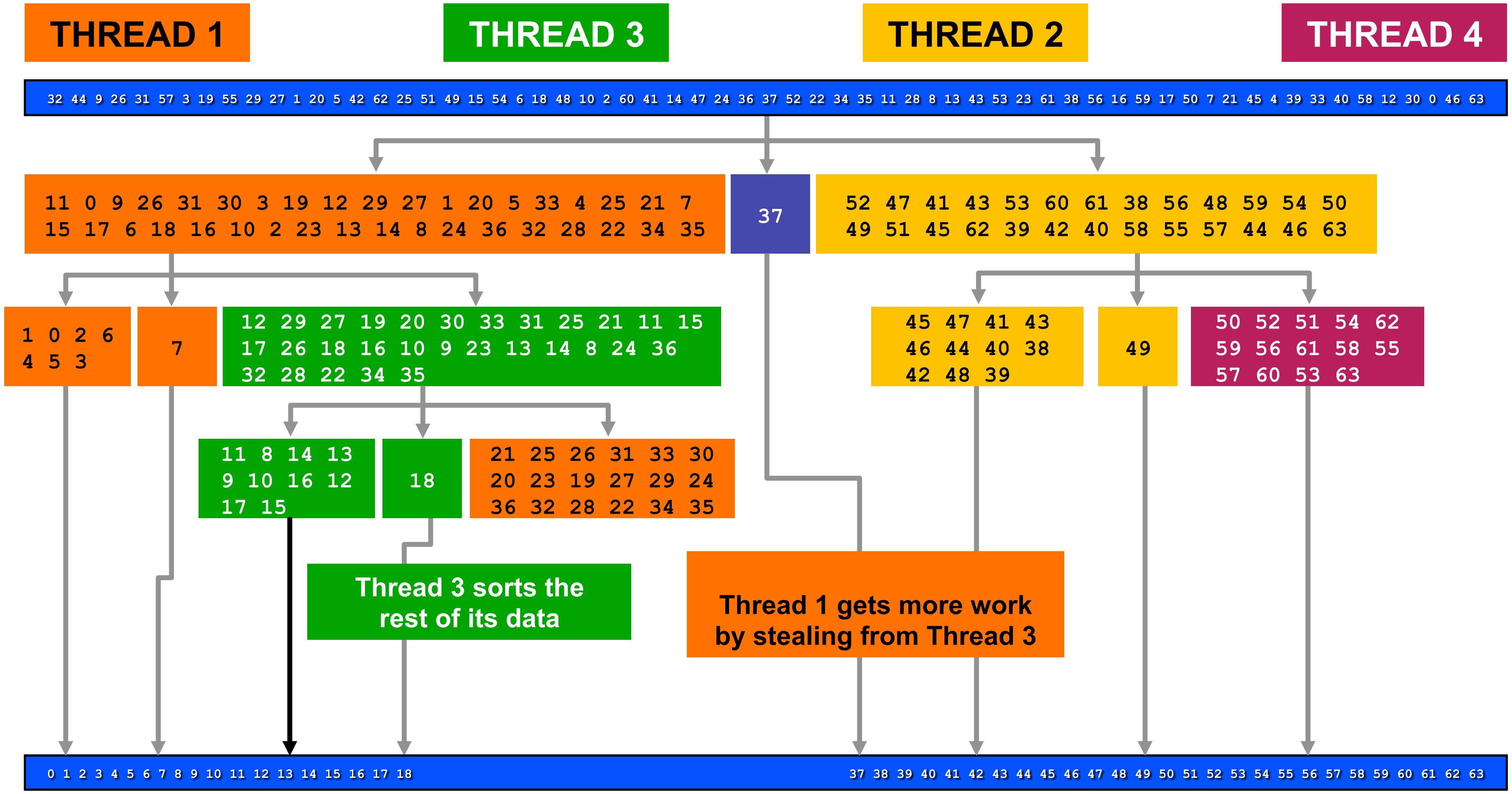
Thread 3 partitions/splits its data

Thread 2 sorts the rest its data

Thread 4 sorts the rest of its data

0 1 2 3 4 5 6 7 18 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63

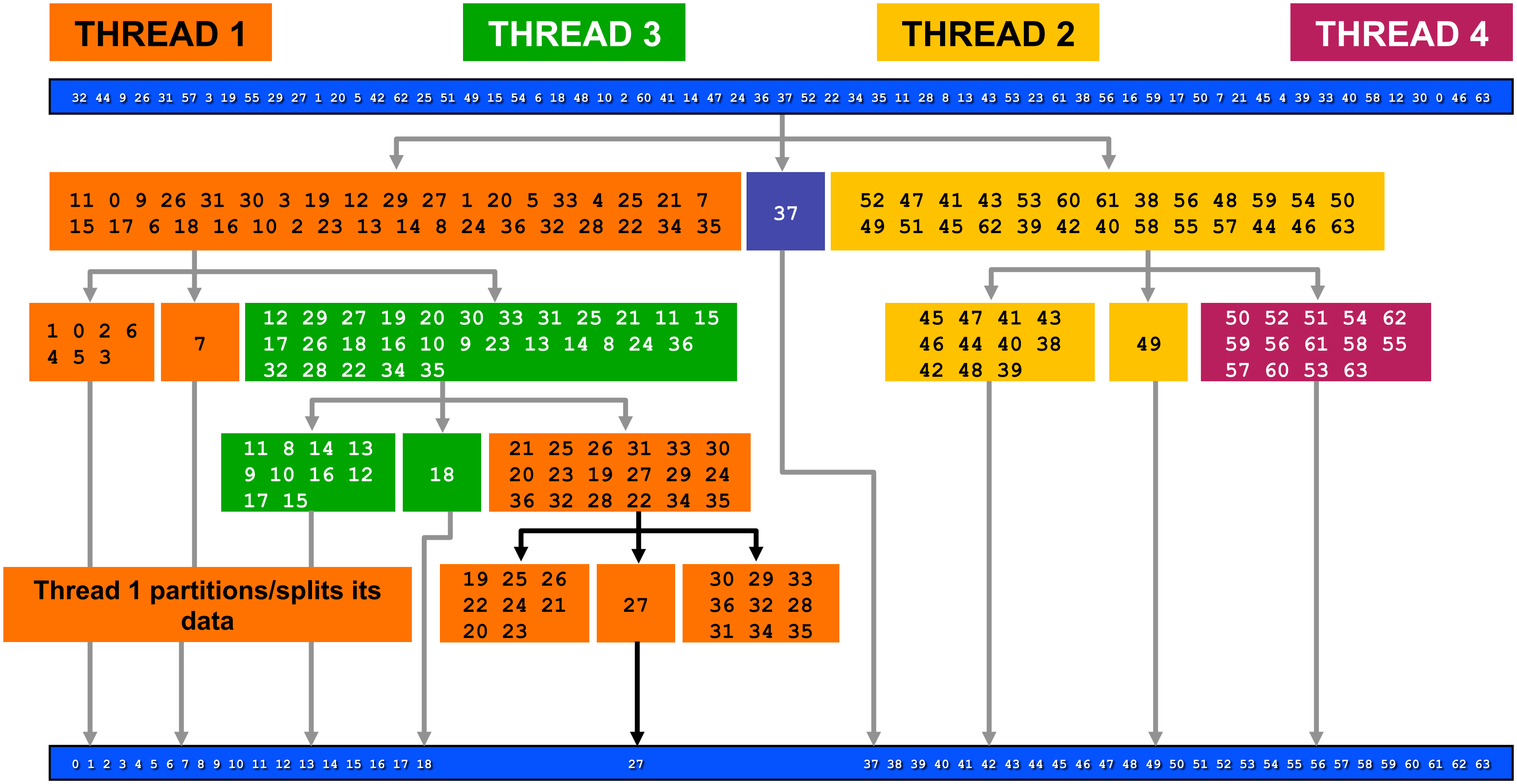
Quicksort – Step 5



Thread 3 sorts the rest of its data

Thread 1 gets more work by stealing from Thread 3

Quicksort – Step 6



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

1 0 2 6 4 5 3

7

12 29 27 19 20 30 33 31 25 21 11 15 17 26 18 16 10 9 23 13 14 8 24 36 32 28 22 34 35

45 47 41 43 46 44 40 38 42 48 39

49

50 52 51 54 62 59 56 61 58 55 57 60 53 63

11 8 14 13 9 10 16 12 17 15

18

21 25 26 31 33 30 20 23 19 27 29 24 36 32 28 22 34 35

Thread 1 partitions/splits its data

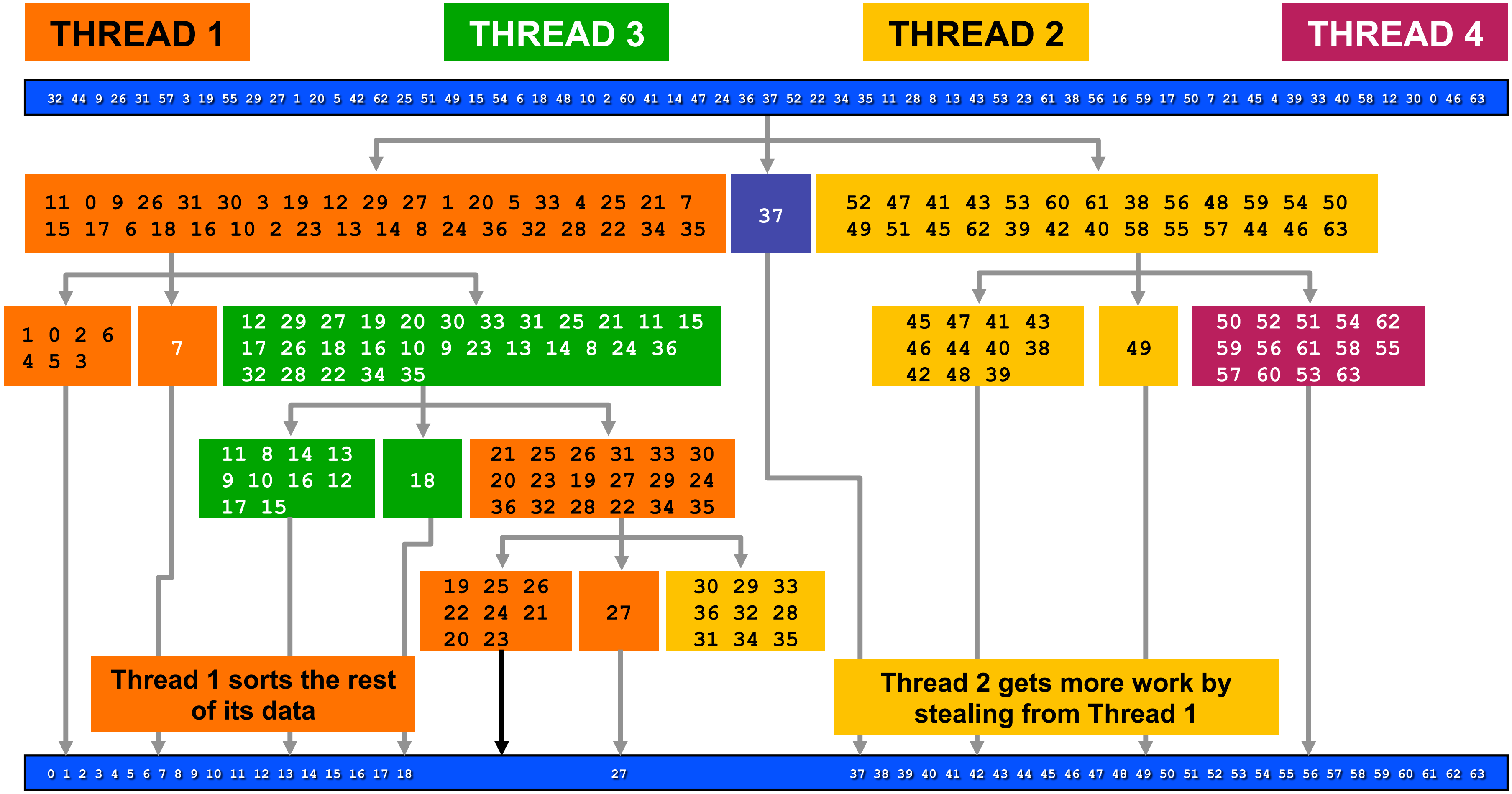
19 25 26 22 24 21 20 23

27

30 29 33 36 32 28 31 34 35

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 27 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63

Quicksort – Step 6



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

1 0 2 6
4 5 3

7

12 29 27 19 20 30 33 31 25 21 11 15
17 26 18 16 10 9 23 13 14 8 24 36
32 28 22 34 35

45 47 41 43
46 44 40 38
42 48 39

49

50 52 51 54 62
59 56 61 58 55
57 60 53 63

11 8 14 13
9 10 16 12
17 15

18

21 25 26 31 33 30
20 23 19 27 29 24
36 32 28 22 34 35

19 25 26
22 24 21
20 23

27

30 29 33
36 32 28
31 34 35

Thread 1 sorts the rest of its data

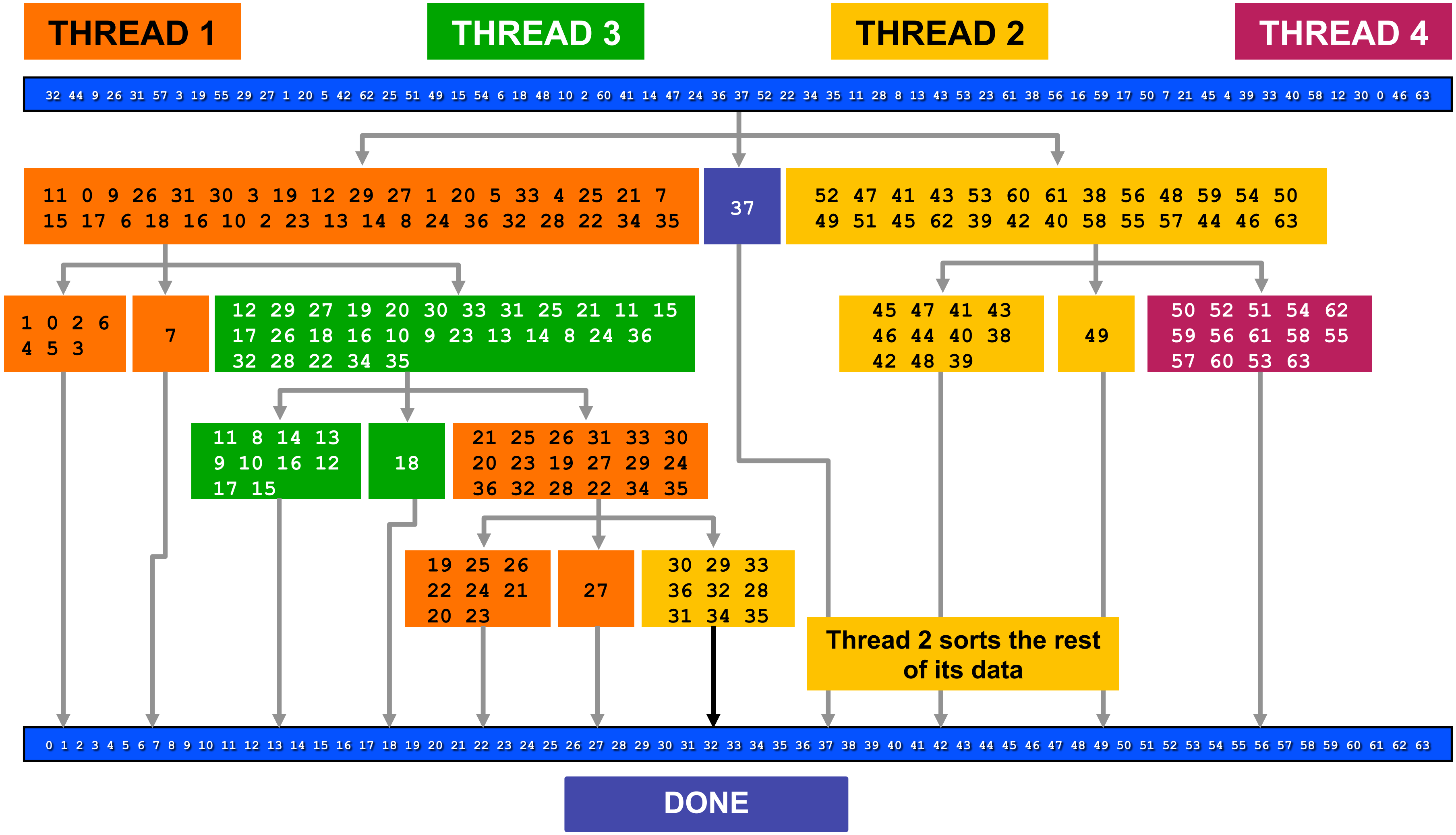
Thread 2 gets more work by stealing from Thread 1

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18

27

37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63

Quicksort – Step 7



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

1 0 2 6 4 5 3

7

12 29 27 19 20 30 33 31 25 21 11 15 17 26 18 16 10 9 23 13 14 8 24 36 32 28 22 34 35

45 47 41 43 46 44 40 38 42 48 39

49

50 52 51 54 62 59 56 61 58 55 57 60 53 63

11 8 14 13 9 10 16 12 17 15

18

21 25 26 31 33 30 20 23 19 27 29 24 36 32 28 22 34 35

19 25 26 22 24 21 20 23

27

30 29 33 36 32 28 31 34 35

Thread 2 sorts the rest of its data

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

DONE

An Example using parallel_for (1 of 3)

- Independent iterations and fixed/known bounds

```
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 3)

- 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;
}
```

Include Library Headers

Use namespace

Initialize scheduler

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

An Example using parallel_for (3 of 3)

- 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());
}
```

Define Task

Use algorithm

Use auto_partitioner()

An Example using parallel_for (3b of 3)

- 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());
}
```

An Example using parallel_for with C++0x lambda functions

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

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

Use lambda function to implement MyBody::operator() inside the call to parallel_for().

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

Closer resemblance to sequential code

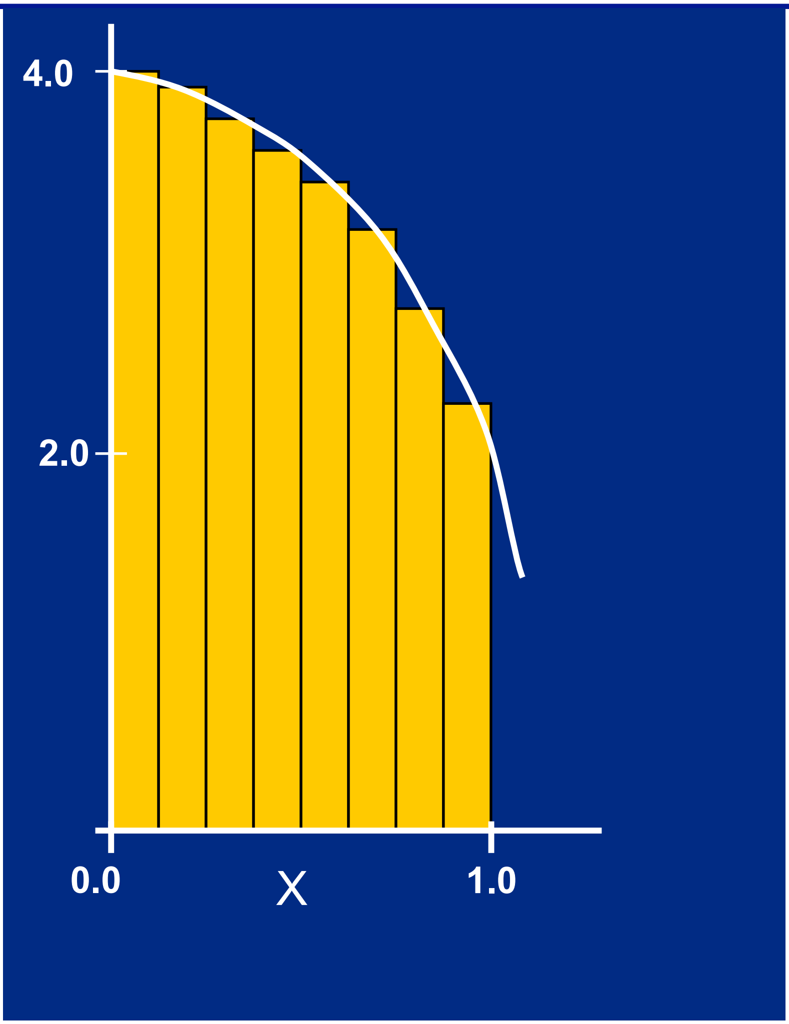
The parallel_reduce Template

```
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



```
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

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

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

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.

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

BERKELEY PAR LAB

Lithe: Enabling Efficient Composition of Parallel Libraries

Heidi Pan, Benjamin Hindman, Krste Asanović

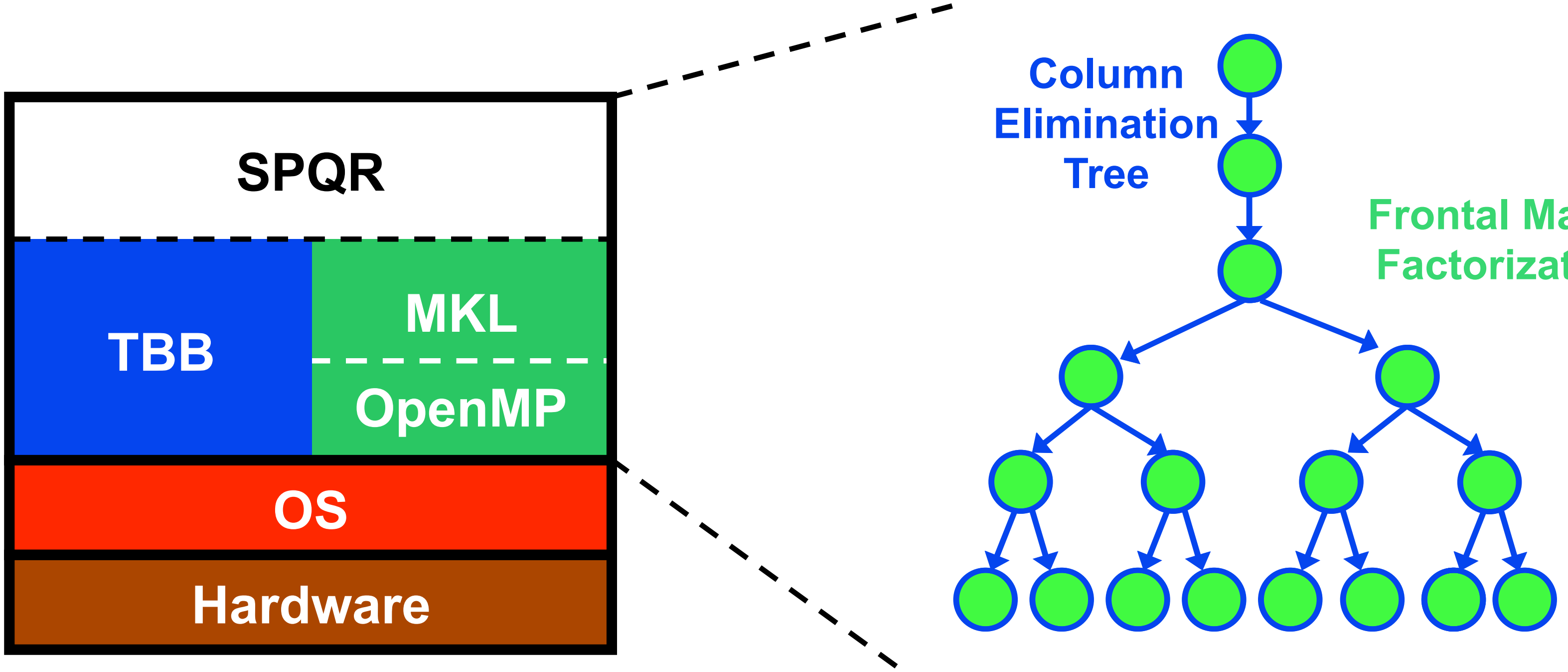
xoxo@mit.edu • {benh, krste}@eecs.berkeley.edu
Massachusetts Institute of Technology • UC Berkeley

ParLab Boot Camp • August 19, 2009

Real-World Parallel Composition Example

Sparse QR Factorization

(Tim Davis, Univ of Florida)

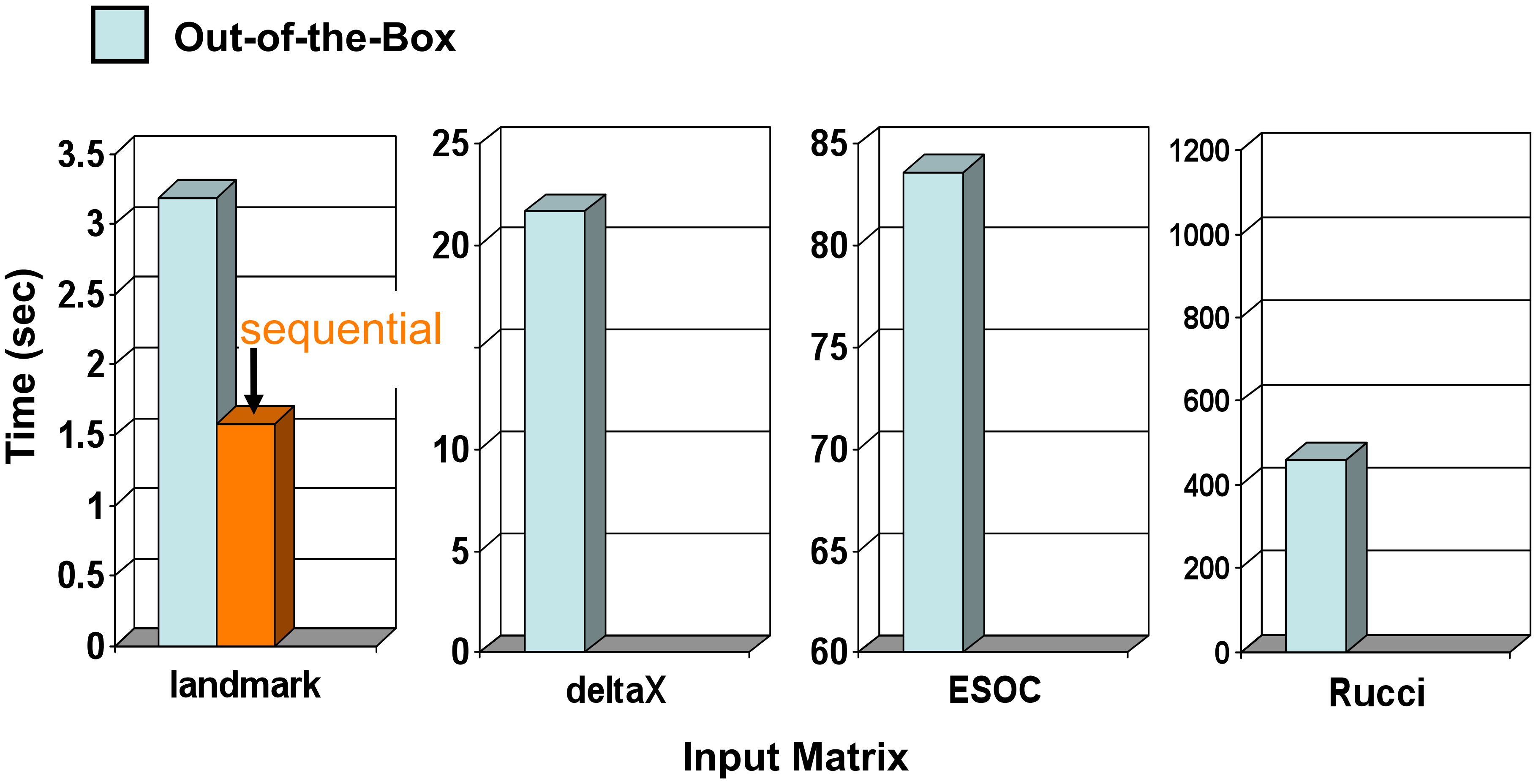


System Stack

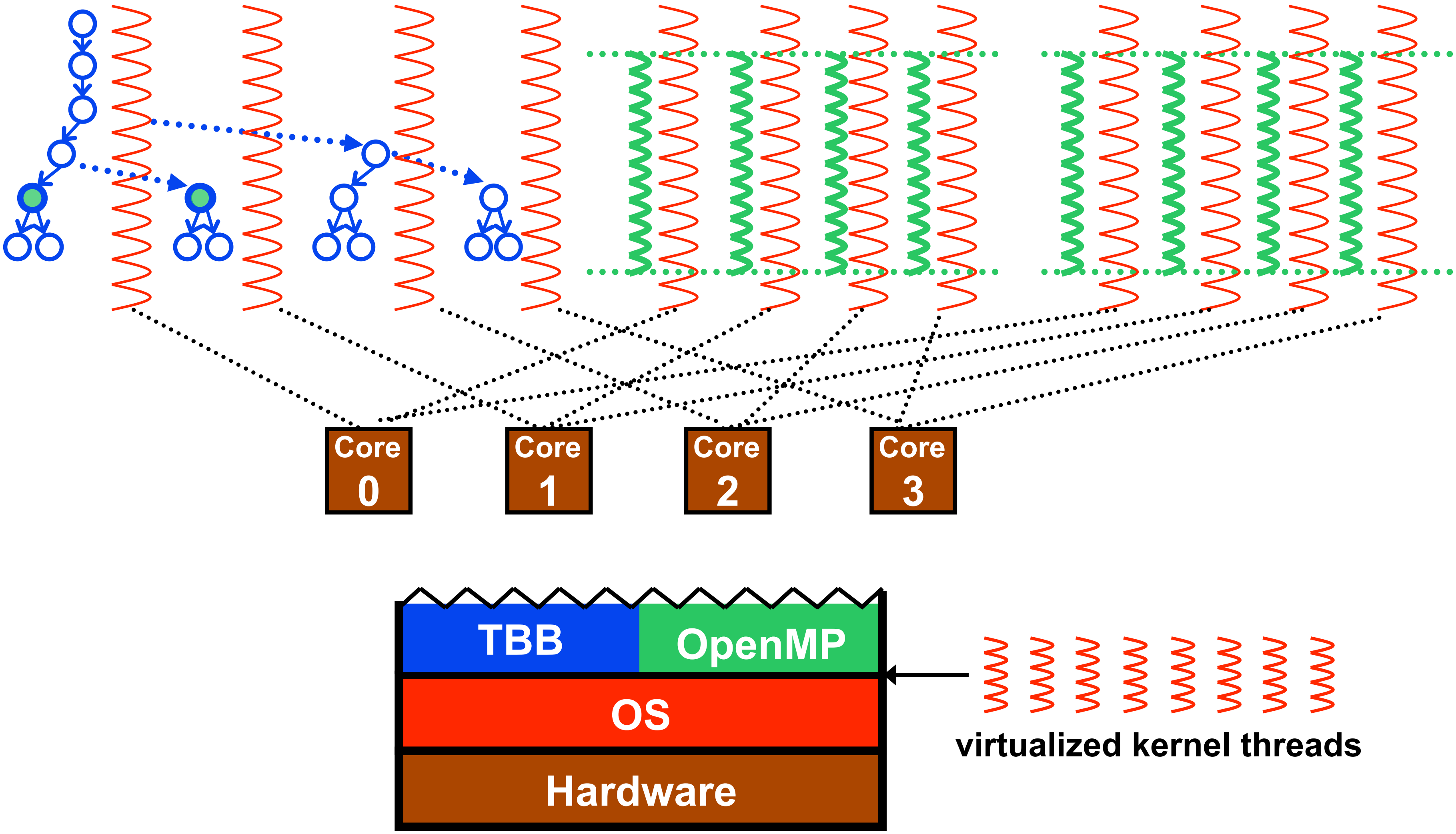
Software Architecture

Out-of-the-Box Performance

Performance of SPQR on 16-core Machine



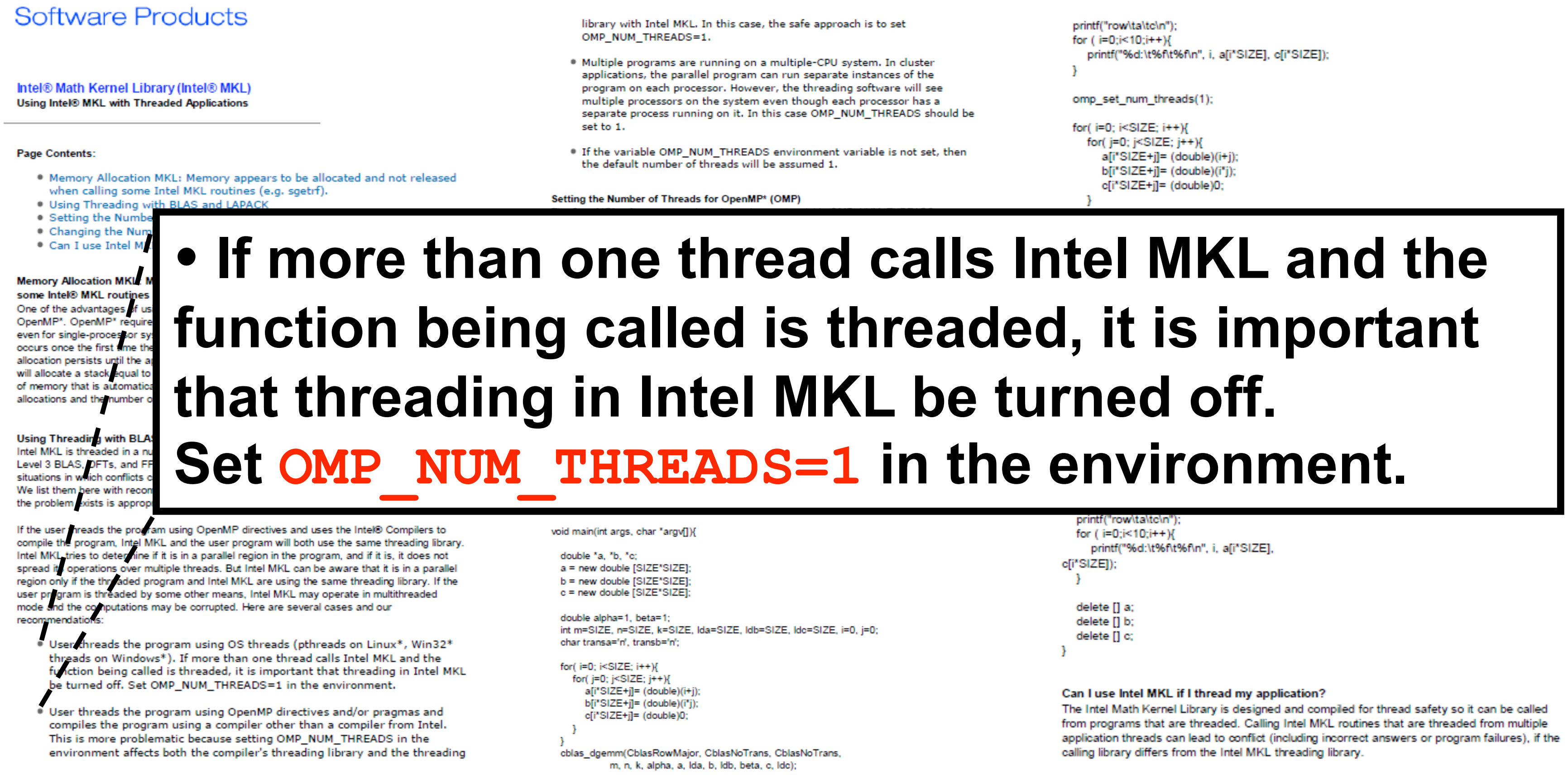
Out-of-the-Box Libraries Oversubscribe the Resources



MKL Quick Fix

Using Intel MKL with Threaded Applications

http://www.intel.com/support/performancetools/libraries/mkl/sb/CS-017177.htm



Software Products

Intel® Math Kernel Library (Intel® MKL)
Using Intel® MKL with Threaded Applications

Page Contents:

- Memory Allocation MKL: Memory appears to be allocated and not released when calling some Intel MKL routines (e.g. sgetrf).
- Using Threading with BLAS and LAPACK
- Setting the Numbe
- Changing the Num
- Can I use Intel M

Memory Allocation MKL: M
some Intel® MKL routines
One of the advantages of usi
OpenMP*. OpenMP* require
even for single-processor sy
occurs once the first time the
allocation persists until the a
will allocate a stack equal to
of memory that is automatica
allocations and the number o

Using Threading with BLAS
Intel MKL is threaded in a nu
Level 3 BLAS, DFTs, and FF
situations in which conflicts c
We list them here with recom
the problem exists is approp

If the user threads the program using OpenMP directives and uses the Intel® Compilers to compile the program, Intel MKL and the user program will both use the same threading library. Intel MKL tries to determine if it is in a parallel region in the program, and if it is, it does not spread its operations over multiple threads. But Intel MKL can be aware that it is in a parallel region only if the threaded program and Intel MKL are using the same threading library. If the user program is threaded by some other means, Intel MKL may operate in multithreaded mode and the computations may be corrupted. Here are several cases and our recommendations:

- User threads the program using OS threads (pthreads on Linux*, Win32* threads on Windows*). If more than one thread calls Intel MKL and the function being called is threaded, it is important that threading in Intel MKL be turned off. Set OMP_NUM_THREADS=1 in the environment.
- User threads the program using OpenMP directives and/or pragmas and compiles the program using a compiler other than a compiler from Intel. This is more problematic because setting OMP_NUM_THREADS in the environment affects both the compiler's threading library and the threading library with Intel MKL. In this case, the safe approach is to set OMP_NUM_THREADS=1.
- Multiple programs are running on a multiple-CPU system. In cluster applications, the parallel program can run separate instances of the program on each processor. However, the threading software will see multiple processors on the system even though each processor has a separate process running on it. In this case OMP_NUM_THREADS should be set to 1.
- If the variable OMP_NUM_THREADS environment variable is not set, then the default number of threads will be assumed 1.

Setting the Number of Threads for OpenMP* (OMP)

```
void main(int args, char *argv[]){

  double *a, *b, *c;
  a = new double [SIZE*SIZE];
  b = new double [SIZE*SIZE];
  c = new double [SIZE*SIZE];

  double alpha=1, beta=1;
  int m=SIZE, n=SIZE, k=SIZE, lda=SIZE, ldb=SIZE, ldc=SIZE, i=0, j=0;
  char transa='n', transb='n';

  for( i=0; i<SIZE; i++){
    for( j=0; j<SIZE; j++){
      a[i*SIZE+j]= (double)(i+j);
      b[i*SIZE+j]= (double)(i*j);
      c[i*SIZE+j]= (double)0;
    }
  }
  cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
              m, n, k, alpha, a, lda, b, ldb, beta, c, ldc);
```

```
printf("row\ta\tc\n");
for ( i=0;i<10;i++){
  printf("%d:\t%f\t%f\n", i, a[i*SIZE], c[i*SIZE]);
}

omp_set_num_threads(1);

for( i=0; i<SIZE; i++){
  for( j=0; j<SIZE; j++){
    a[i*SIZE+j]= (double)(i+j);
    b[i*SIZE+j]= (double)(i*j);
    c[i*SIZE+j]= (double)0;
  }
```

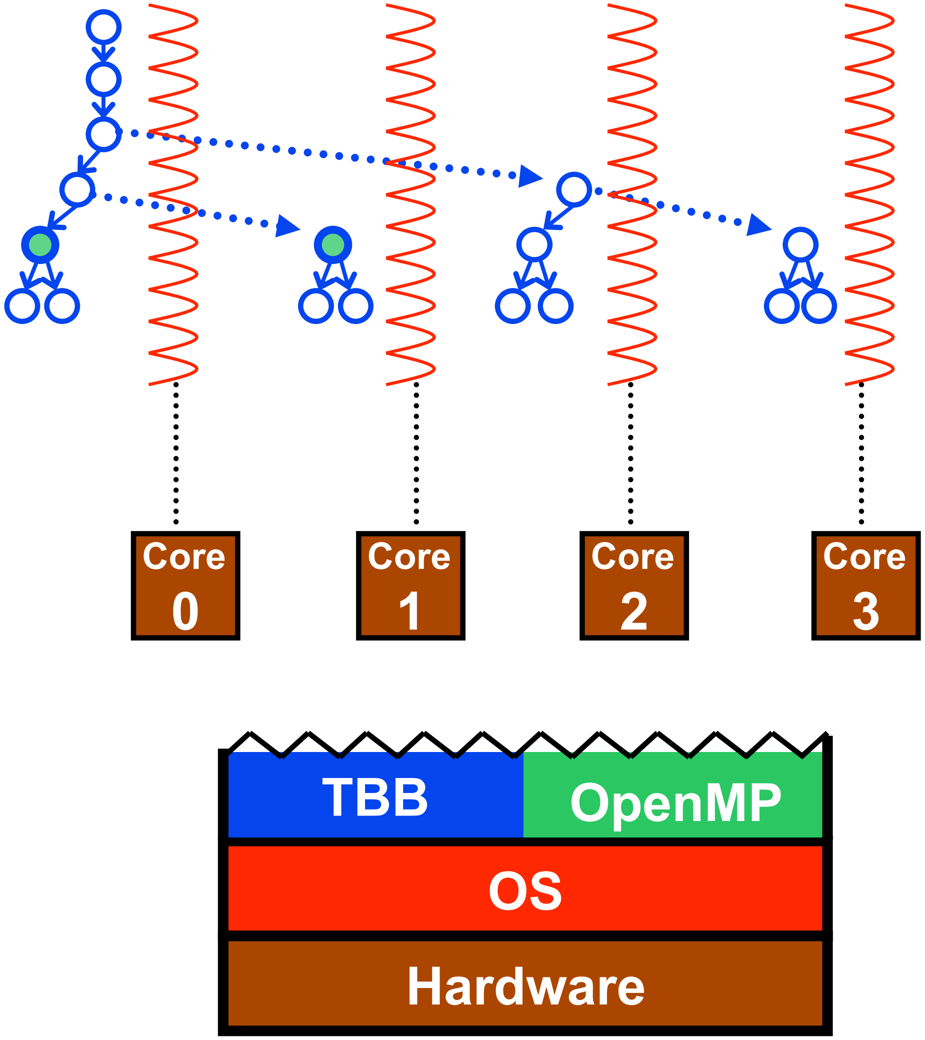
```
  printf("row\ta\tc\n");
  for ( i=0;i<10;i++){
    printf("%d:\t%f\t%f\n", i, a[i*SIZE],
c[i*SIZE]);
  }

  delete [] a;
  delete [] b;
  delete [] c;
}
```

Can I use Intel MKL if I thread my application?
The Intel Math Kernel Library is designed and compiled for thread safety so it can be called from programs that are threaded. Calling Intel MKL routines that are threaded from multiple application threads can lead to conflict (including incorrect answers or program failures), if the calling library differs from the Intel MKL threading library.

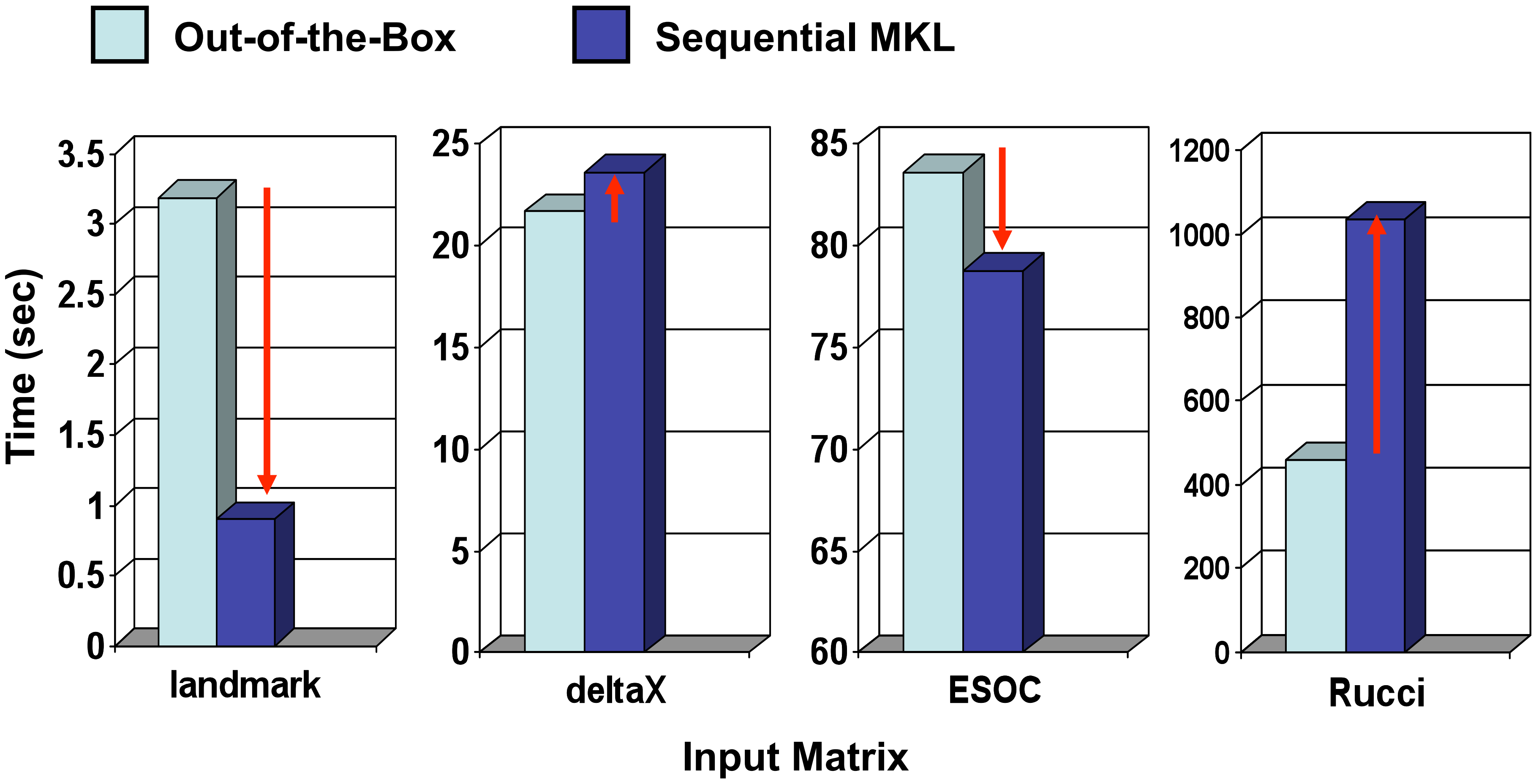
- **If more than one thread calls Intel MKL and the function being called is threaded, it is important that threading in Intel MKL be turned off.**
  Set `OMP_NUM_THREADS=1` in the environment.

Sequential MKL in SPQR

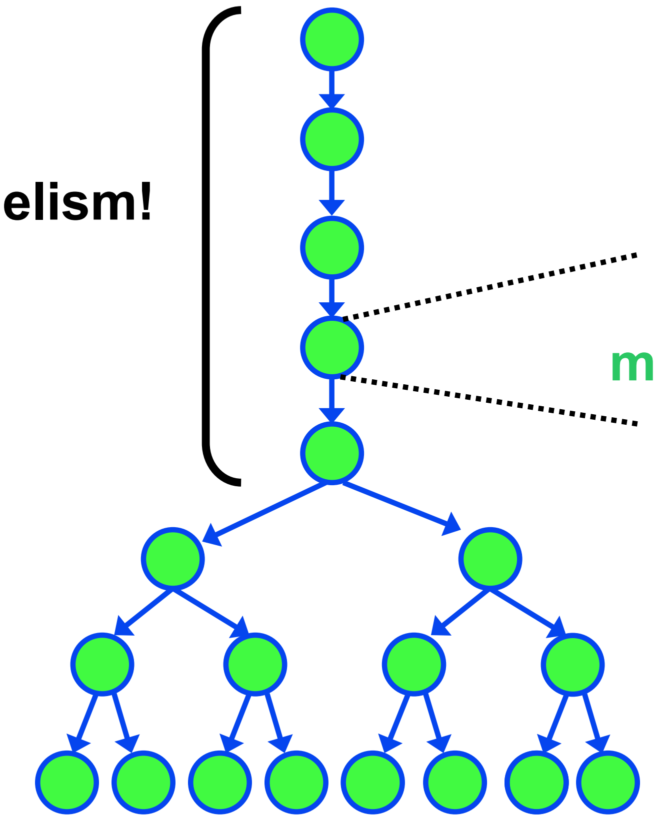


Sequential MKL Performance

Performance of SPQR on 16-core Machine



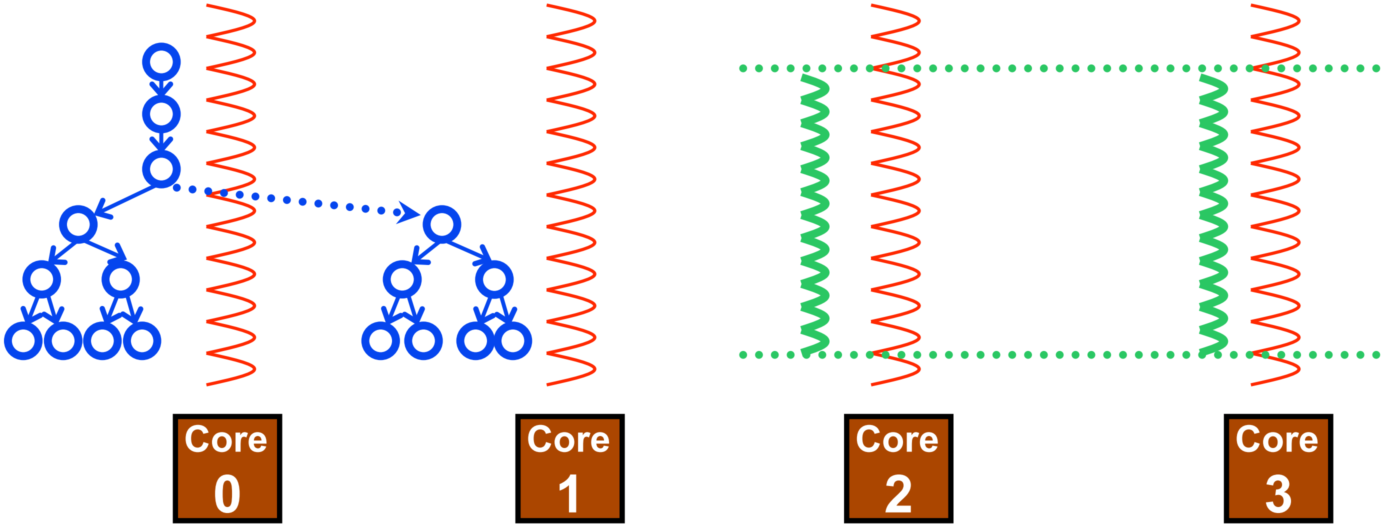
SPQR Wants to Use Parallel MKL



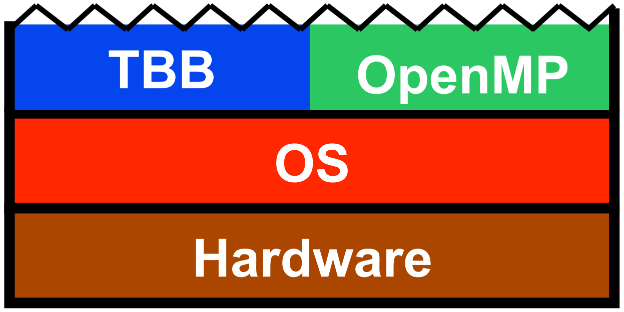
No task-level parallelism!

Want to exploit matrix-level parallelism.

Share Resources Cooperatively



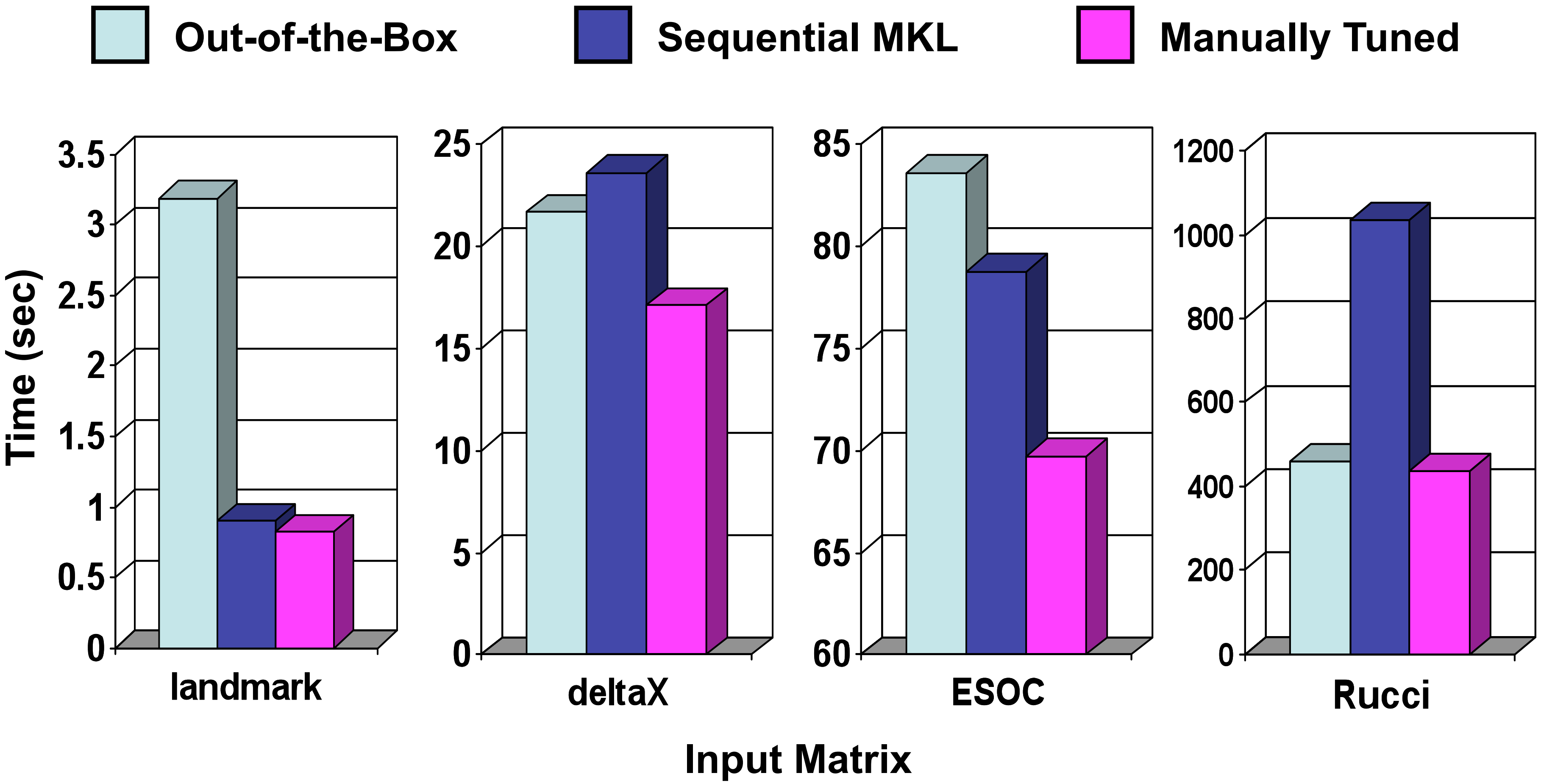
`TBB_NUM_THREADS = 2` `OMP_NUM_THREADS = 2`



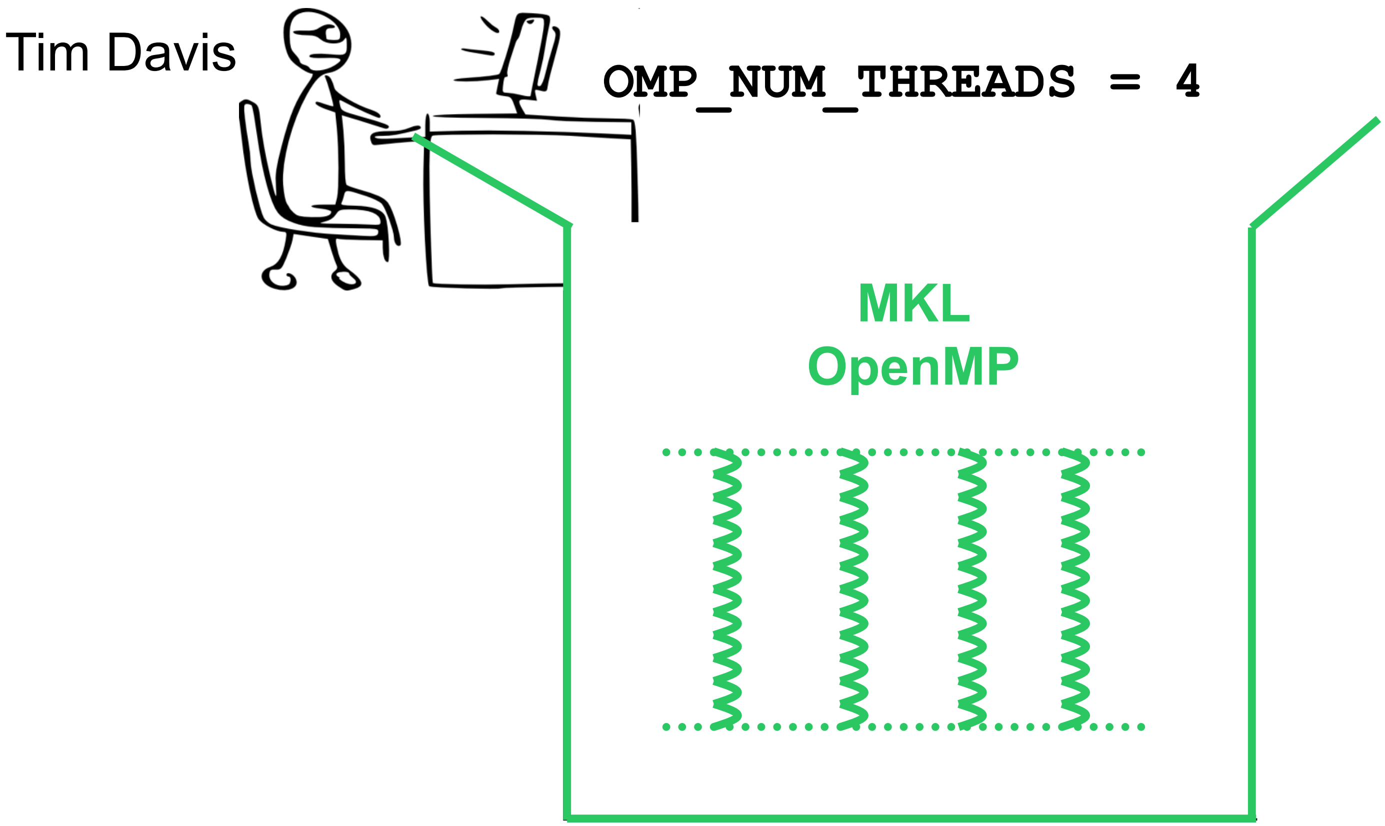
Tim Davis manually tunes libraries to effectively partition the resources.

Manually Tuned Performance

Performance of SPQR on 16-core Machine



Manual Tuning Destroys Black Box Abstractions

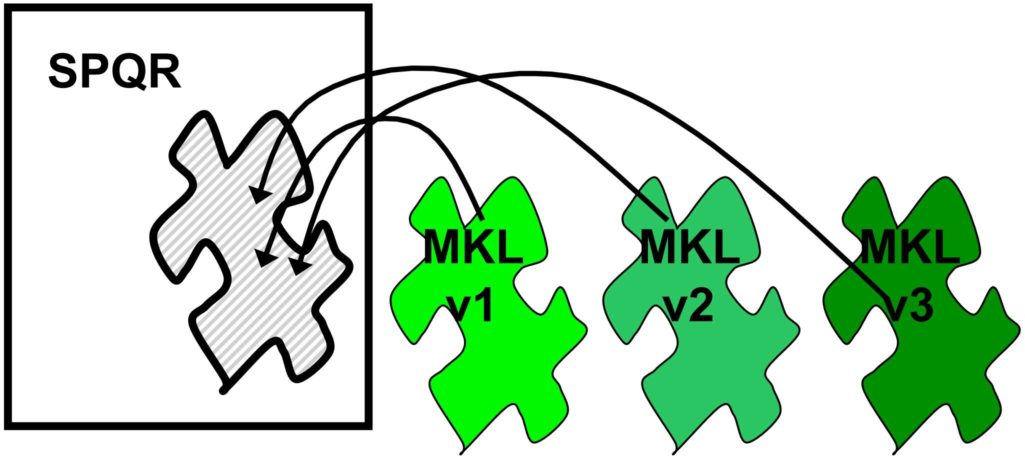


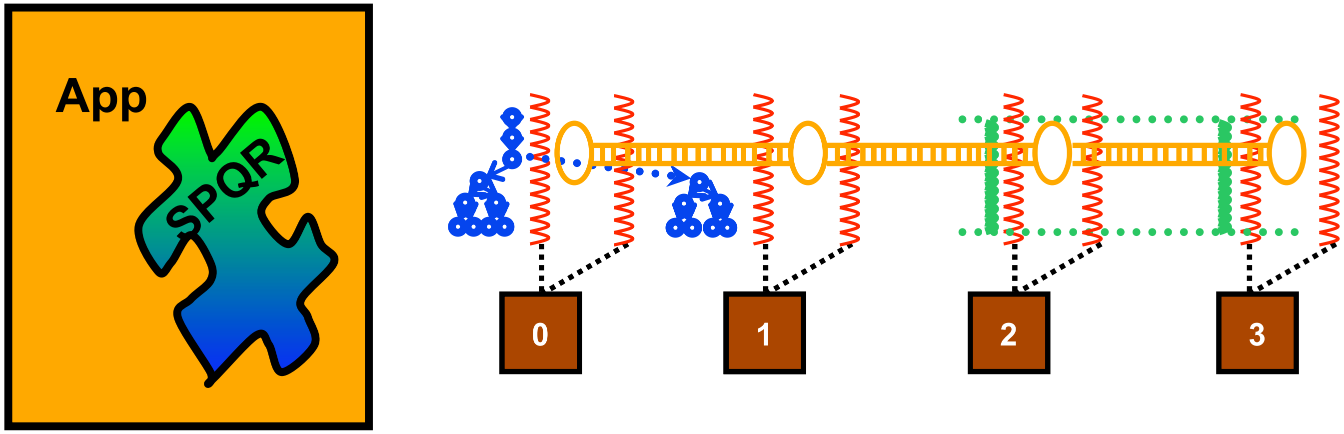
Tim Davis

`OMP_NUM_THREADS = 4`

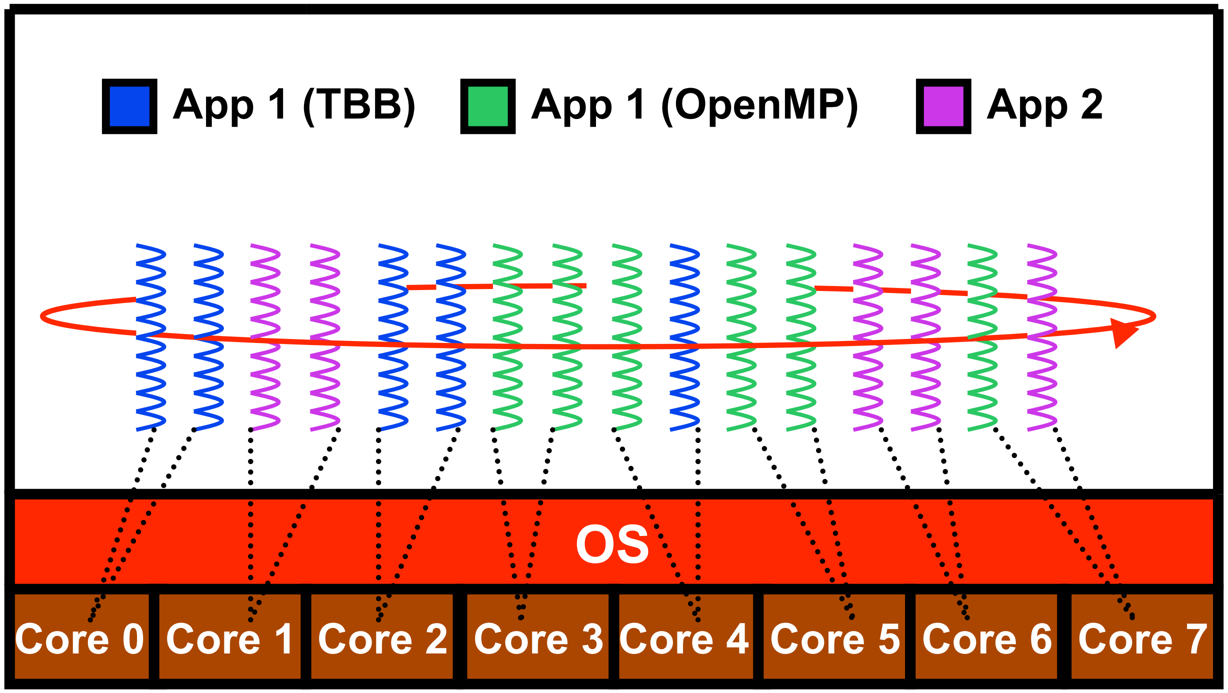
MKL
OpenMP

Manual Tuning Destroys Code Reuse and Modular Updates



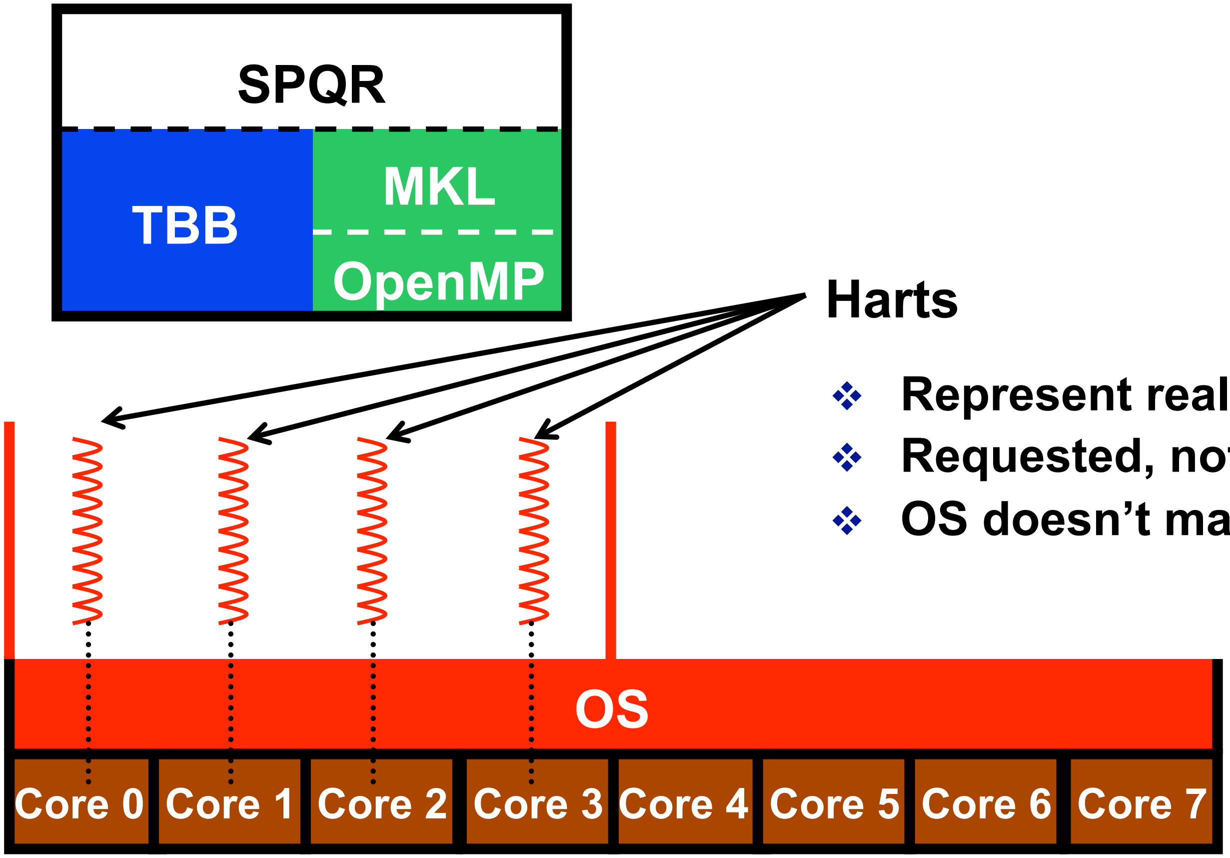


Virtualized Threads are Bad



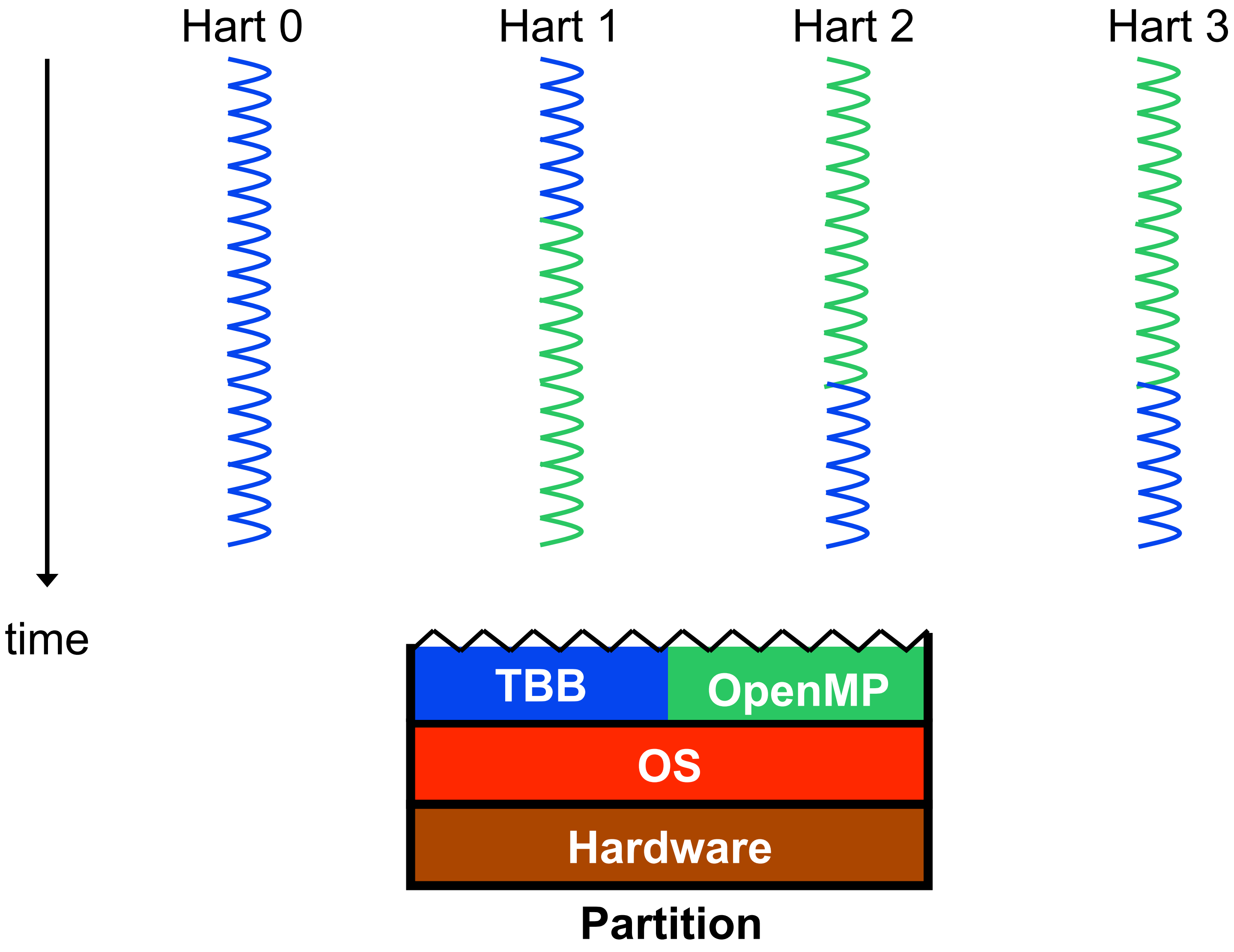
Different codes compete unproductively for resources.

Harts: Hardware Thread Contexts

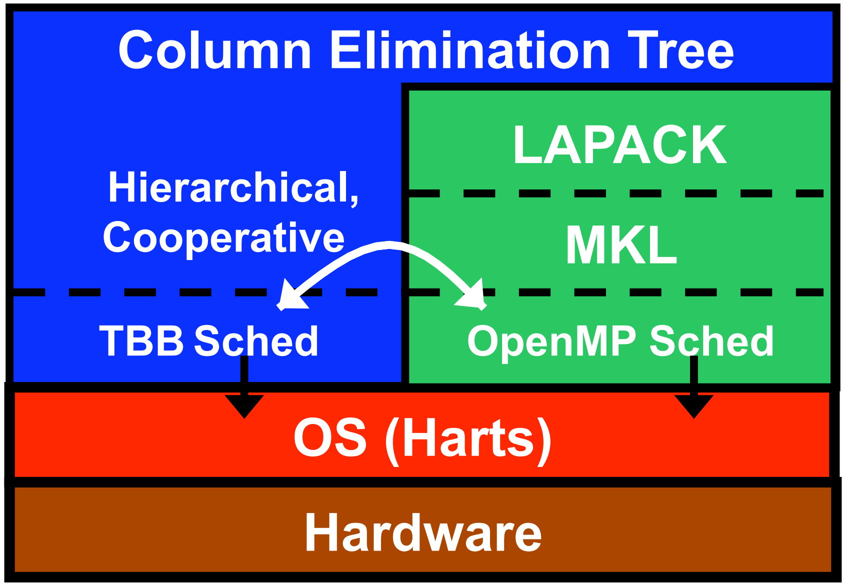


- Represent real hw resources.
- Requested, not created.
- OS doesn’t manage harts for app.

Sharing Harts



Hierarchical Cooperative Scheduling



Column Elimination Tree

LAPACK

Hierarchical, Cooperative

MKL

TBB Sched

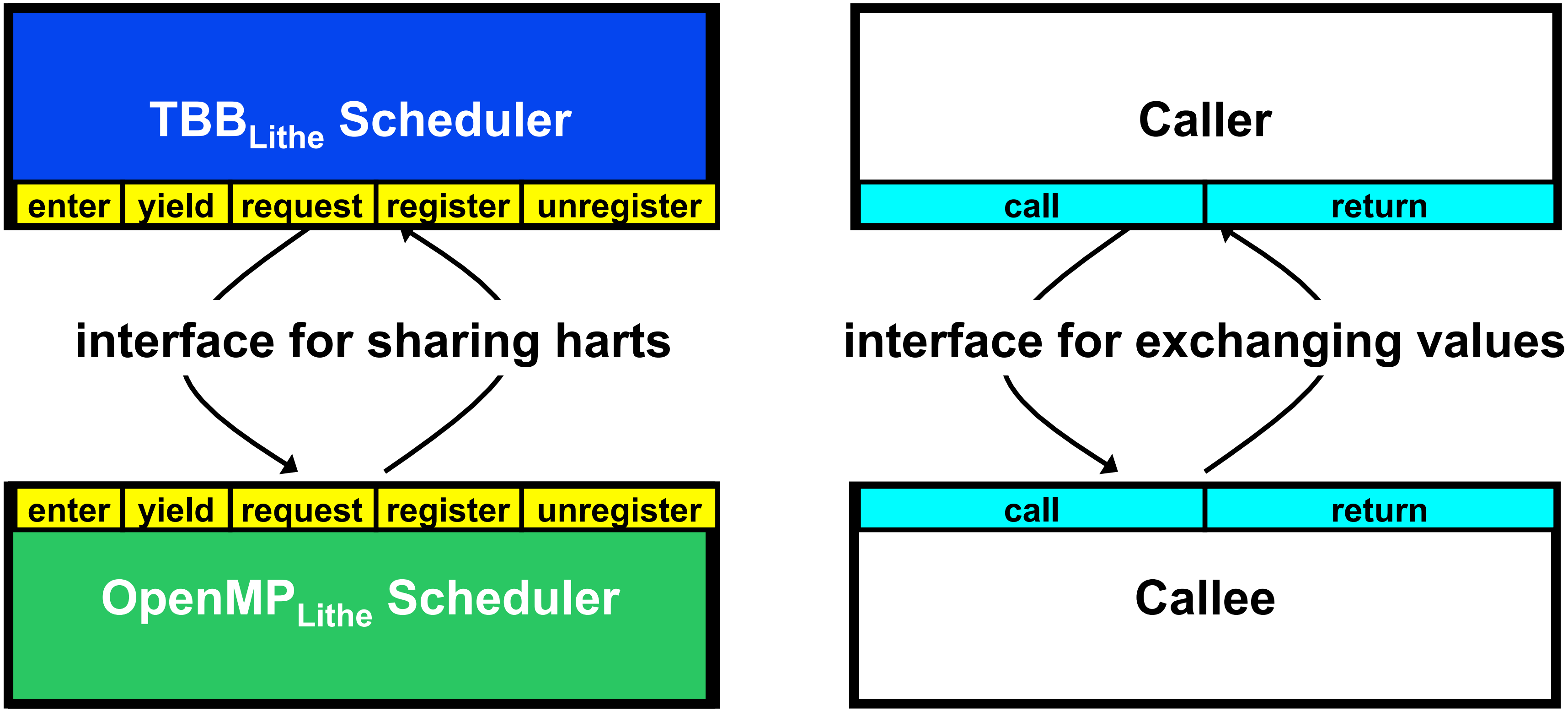
OpenMP Sched

OS (Harts)

Hardware

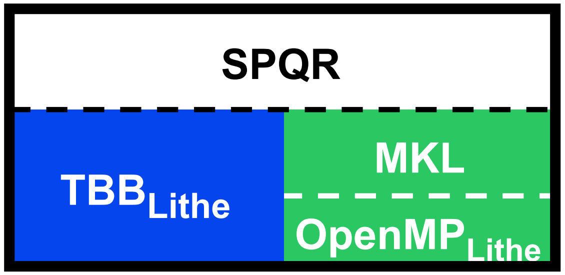
Direct Control of Resources

Standard Lithe ABI



- Analogous to function call ABI for enabling interoperable codes.
- Mechanism for sharing harts, *not* policy.

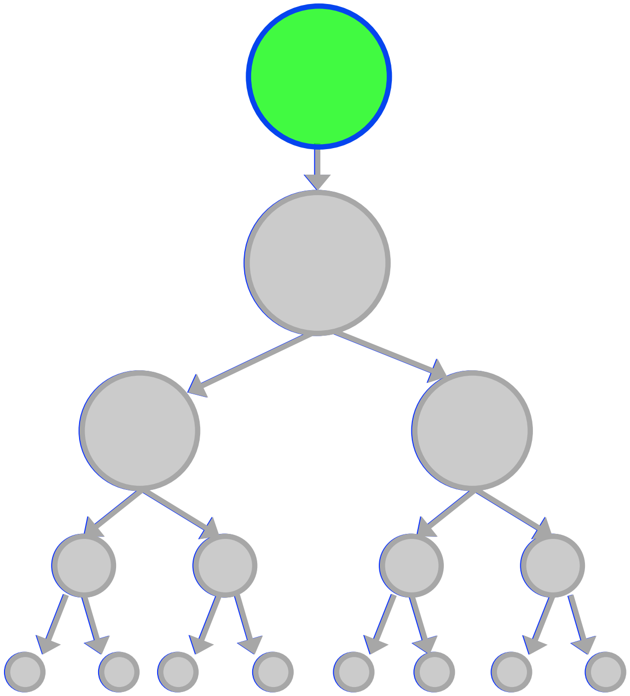
SPQR with Lithe

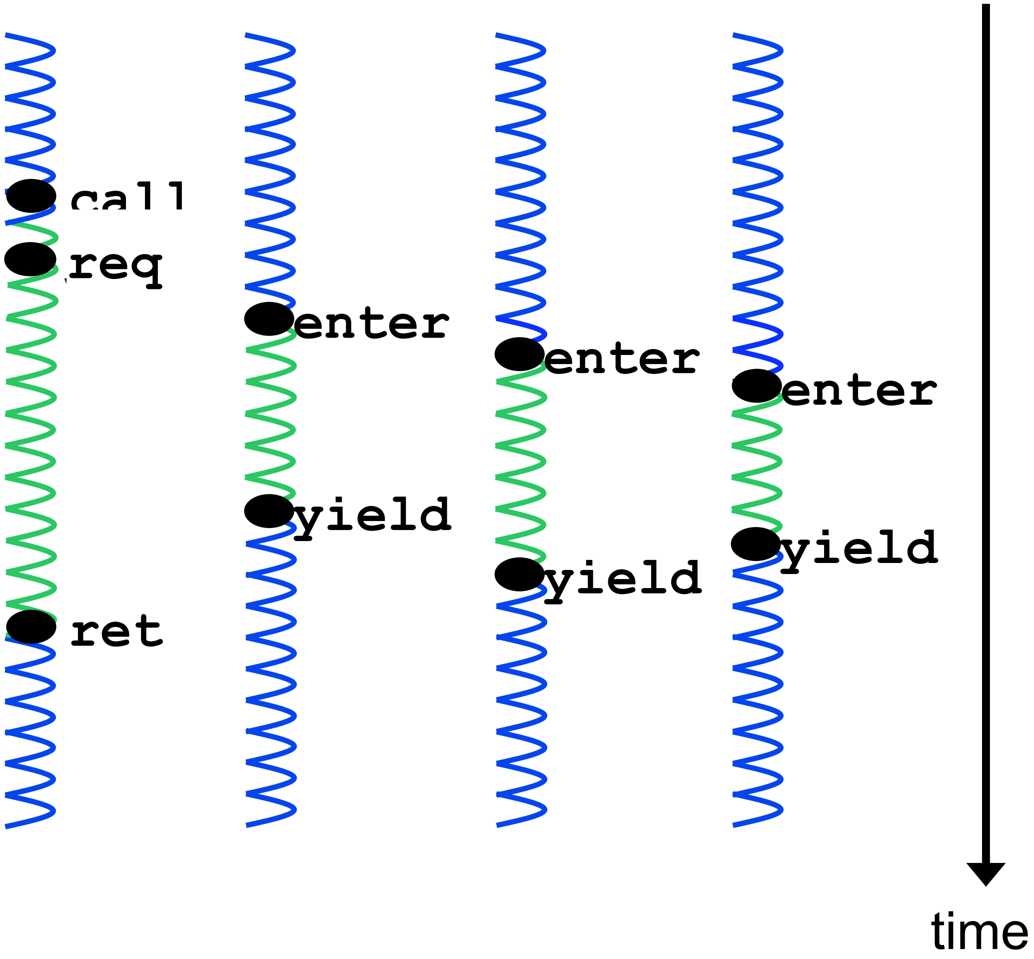

SPQR

$\text{TBB}_{\text{Lithe}}$

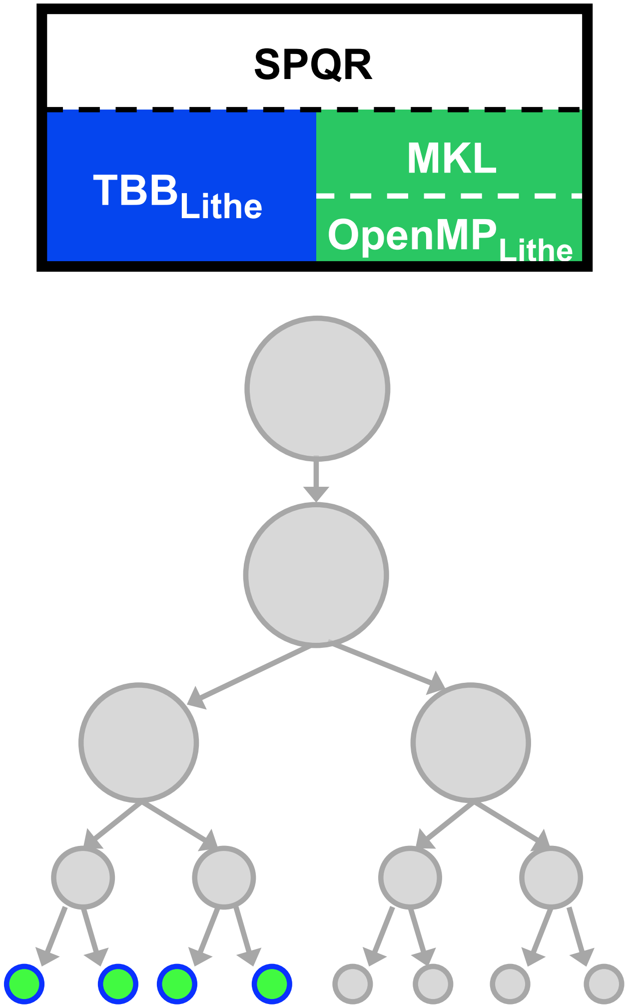
MKL

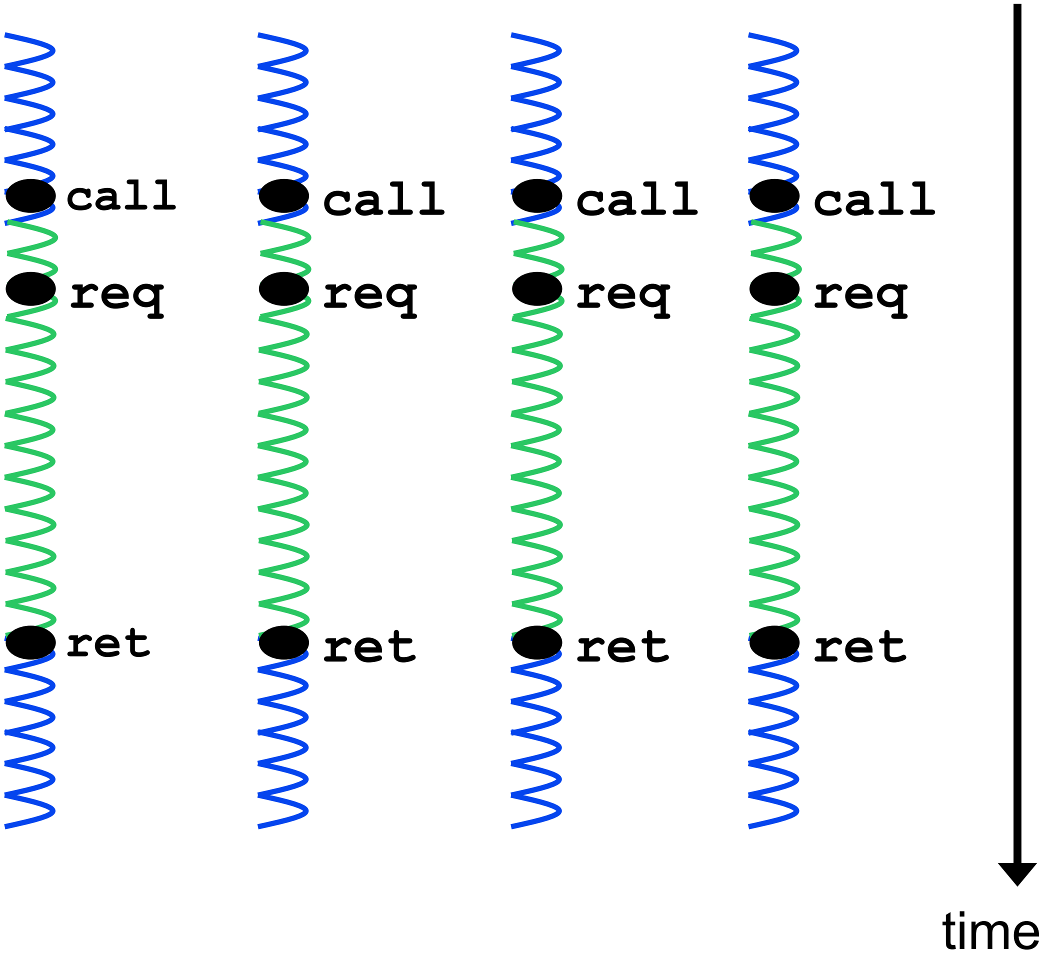
$\text{OpenMP}_{\text{Lithe}}$



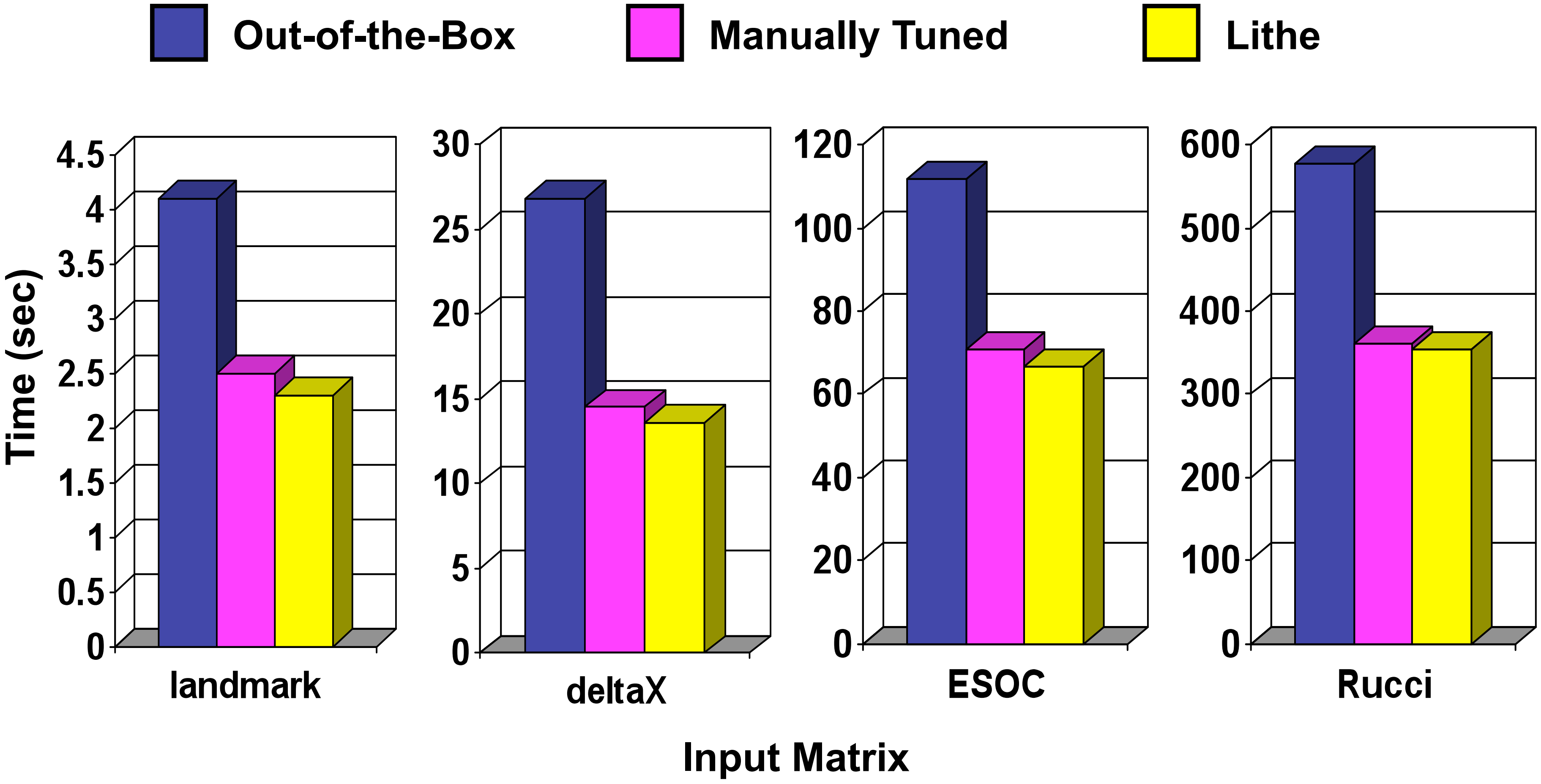


SPQR with Lithe

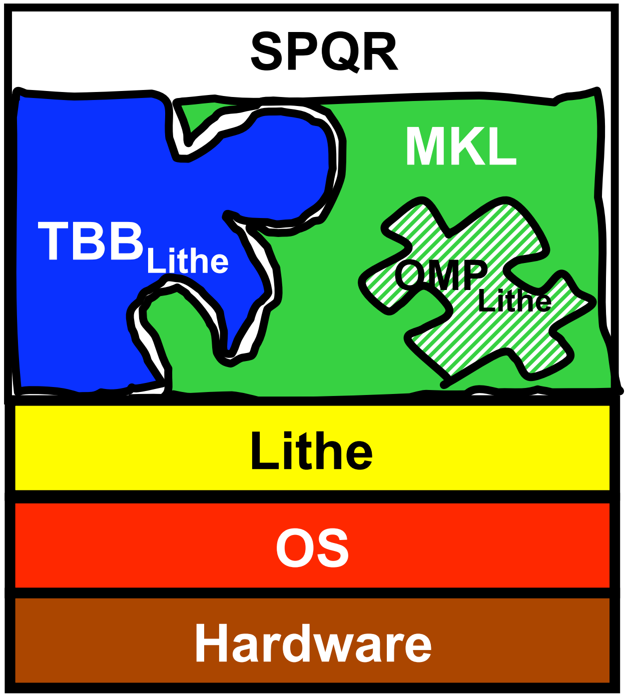




Performance of SPQR with Lithe



Questions?



Lithe: Enabling Efficient Composition of Parallel Libraries

Acknowledgements

We would like to thank George Necula and the rest of Berkeley Par Lab for their feedback on this work.

Research supported by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227). This work has also been in part supported by a National Science Foundation Graduate Research Fellowship. Any opinions, findings, conclusions, or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation. The authors also acknowledge the support of the Gigascale Systems Research Focus Center, one of five research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation program.