

Task parallelism and Intel TBB

(more on shared memory parallelism)

Agenda

1. Introduction. Functional vs task decomposition.
2. Key features of Intel TBB
3. Patterns
4. Task execution and stealing
5. Optimizations
6. Graph of tasks
7. Synchronization
8. Containers

1. Introduction


Two main aspects to parallel programming:

- **Correctness:** avoiding race conditions and deadlock
- **Performance:** efficient use of resources
 - Hardware threads (match parallelism to hardware threads)
 - Memory space (choose right evaluation order)
 - Memory bandwidth (reuse cache)

- Threads are unstructured
- Model doesn't provide scalability


```
pthread_t id[N_THREAD];  
for(int i=0; i<N_THREAD; i++) {  
    int ret = pthread_create(&id[i], NULL, thread_routine, (void*)i);  
    assert(ret==0);  
}
```

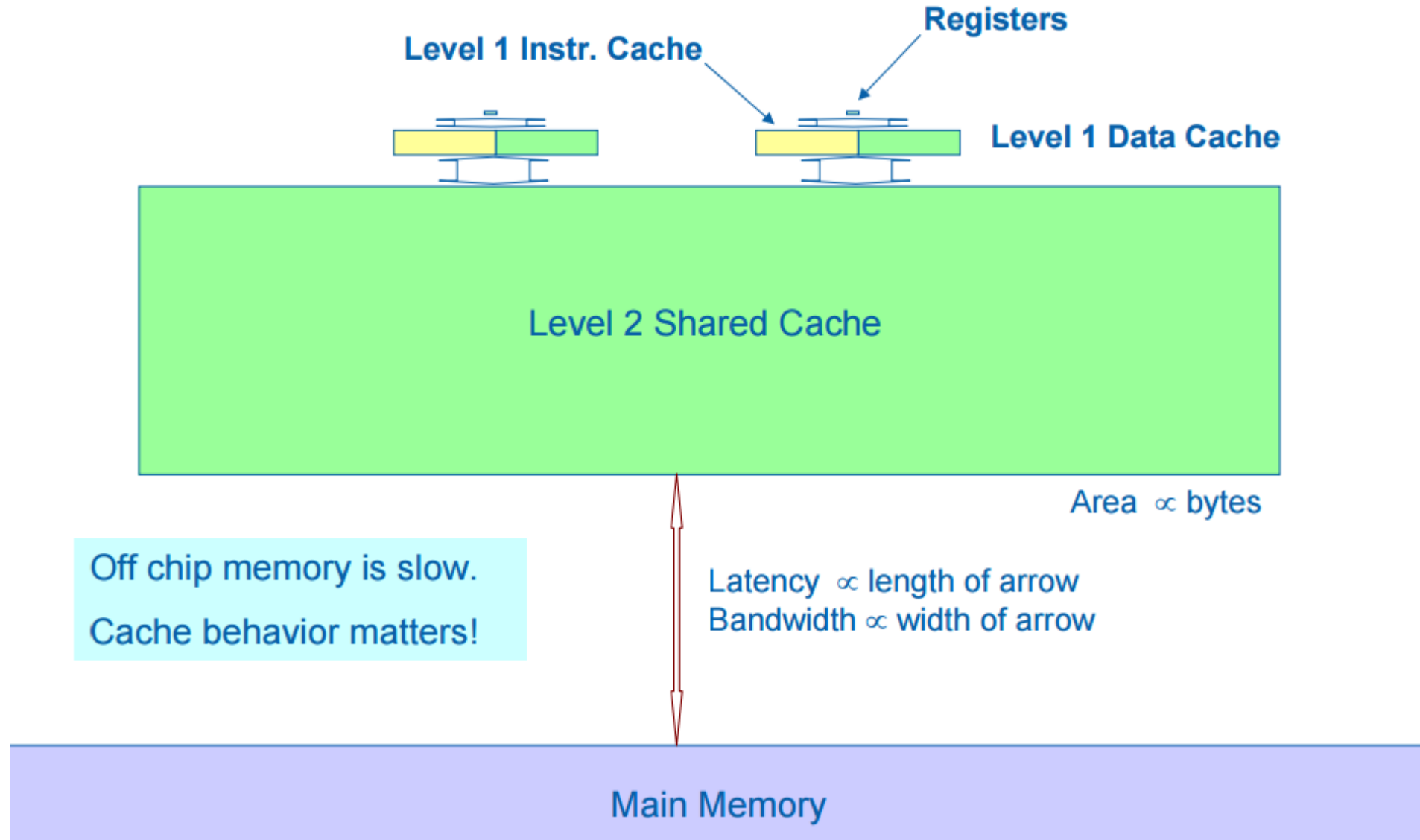
parallel goto



```
for( int i = 0; i < N_THREAD; i++ ) {  
    int ret = pthread_join(id[i], NULL);  
    assert(ret==0);  
}
```

parallel come from





- Locality matters for performance.
- Sieve of Eratosthenes for finding Primes (un-optimized version):
 - Runs out of cache when step is high enough (remember cache line concept)
 - Has to throw cached data at each new iteration.

Start with odd integers

3	5	7	9	11	13	15	17	19	21	23	25	27	29	31	33	35	37	39
---	---	---	---	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

Strike out odd multiples of 3

3	5	7	9	11	13	15	17	19	21	23	25	27	29	31	33	35	37	39
---	---	---	---	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

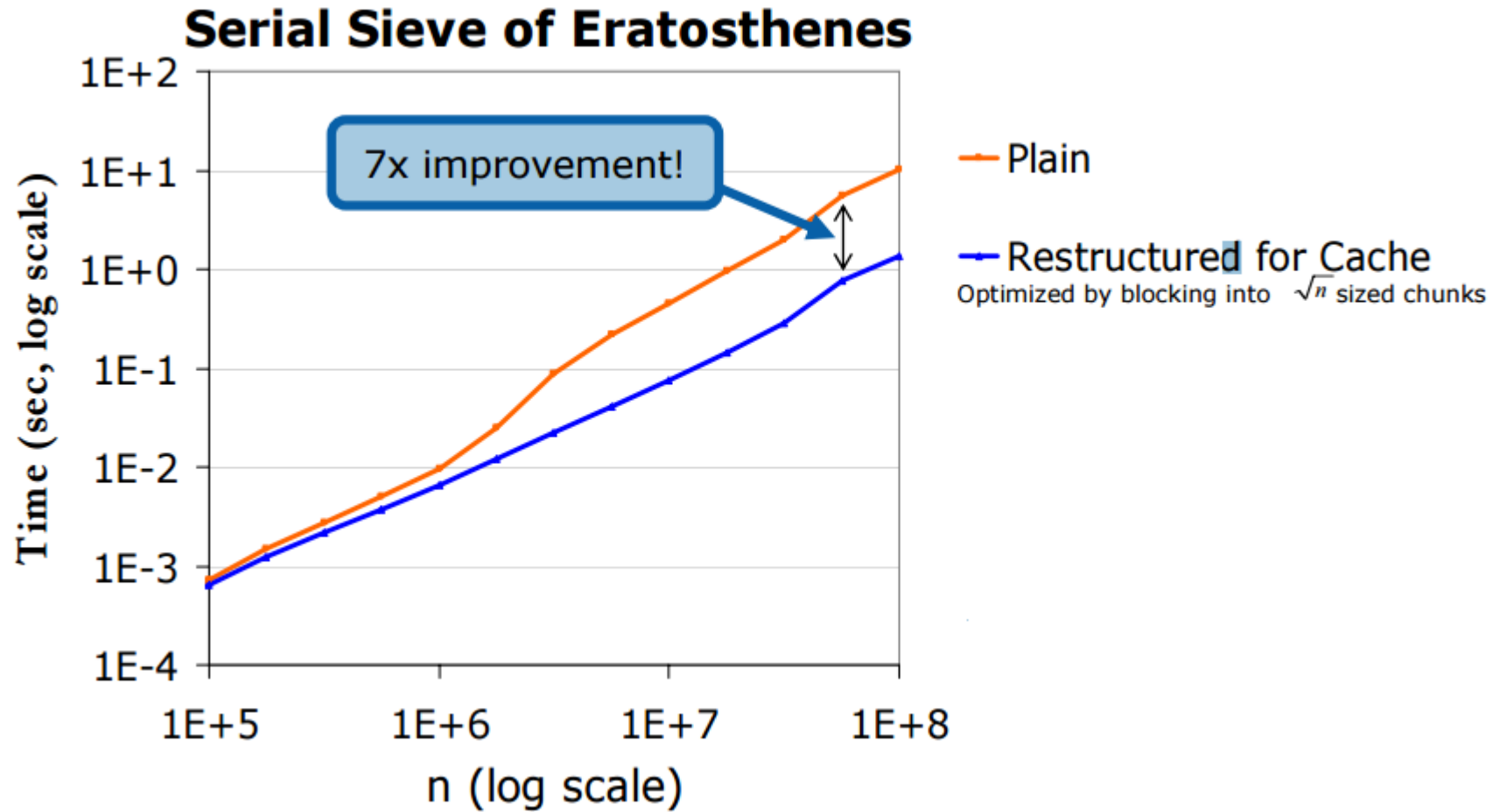
Strike out odd multiples of 5

3	5	7	9	11	13	15	17	19	21	23	25	27	29	31	33	35	37	39
---	---	---	---	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

Strike out odd multiples of 7

3	5	7	9	11	13	15	17	19	21	23	25	27	29	31	33	35	37	39
---	---	---	---	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

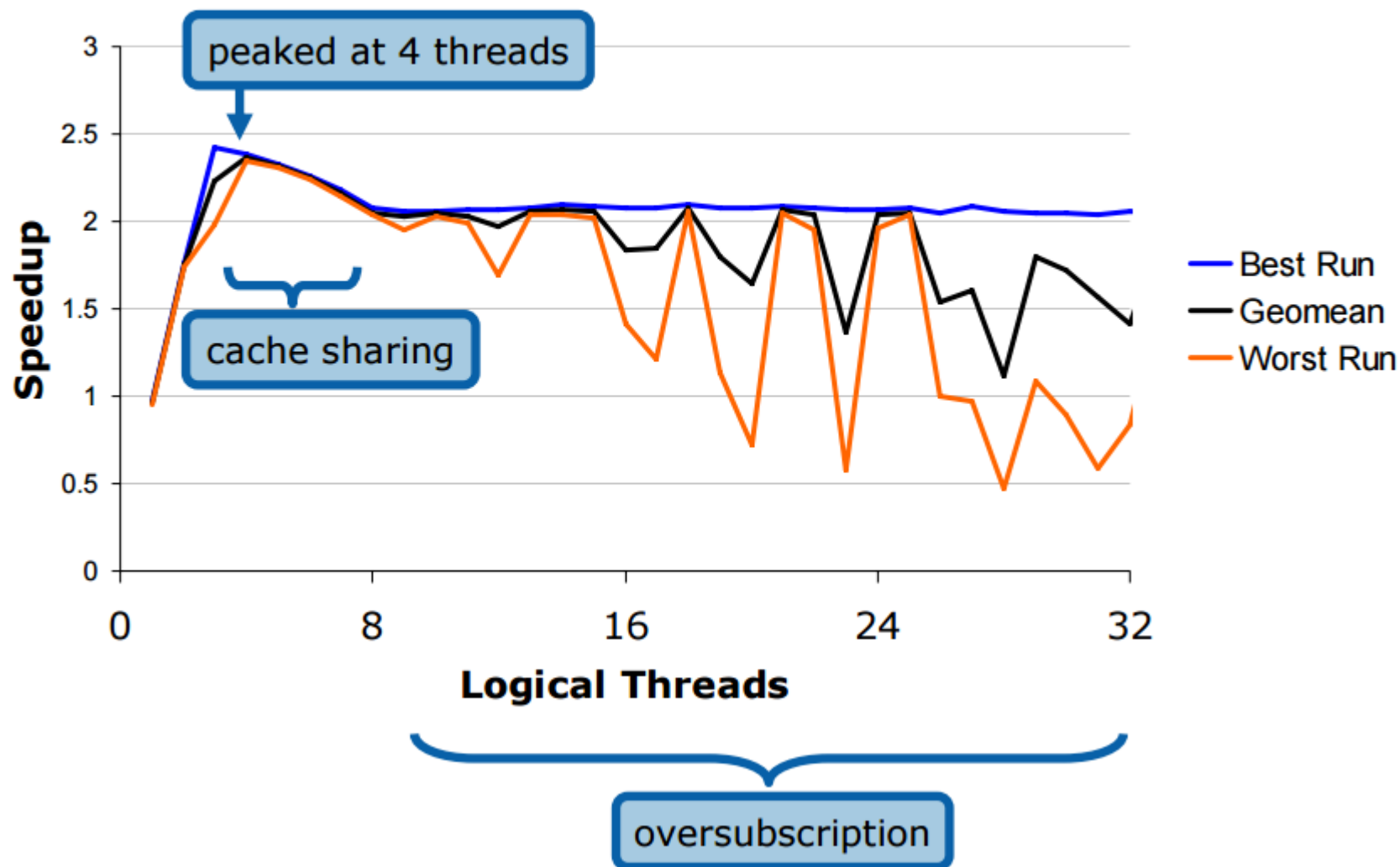
- Optimized version for cache: use \sqrt{n} chunks



- How about nested parallelism ?
- Software components are build from smaller components.
- What if parallelism at each level would be defined by threads ?
 - Remember scheduling concepts from OS
 - How would you control overall performance, fairness and correctitude ?

Effect of Oversubscription

Text filter on 4-socket 8-thread machine with dynamic load balancing



2.Key features of Intel TBB

- Specify **tasks** instead of threads – focus on work not on workers.
 - Library is responsible for managing tasks queue for each worker (show example)
 - Full support for nested parallelism
 - Implements cache and load balancing in the background (e.g task stealing, hottest first).

- Targeted for scalable, data parallel programming
 - Amdahl: fixed fraction of program is serial \Rightarrow speedup limited.
 - Gustafson/Barsis: if dataset grows with number of processors but serial time is fixed \Rightarrow speedup not limited.
 - It is known that functional decomposition (threads) usually do not scale. Impediments:
 - ☐ Coding for a fixed number of threads (can't exploit extra hardware; oversubscribes less hardware)
 - ☐ Contention for shared data (locks cause serialization)

 - TBB approach:
 - Deal with tasks not threads and let runtime optimize their execution
 - Create tasks recursively to improve cache performance
 - Use partial ordering of tasks to avoid the need for locks

Intel® TBB Components

Generic Parallel Algorithms

parallel_for
parallel_while
parallel_reduce
pipeline
parallel_sort
parallel_scan

Concurrent Containers

concurrent_hash_map
concurrent_queue
concurrent_vector

Task scheduler

Synchronization Primitives

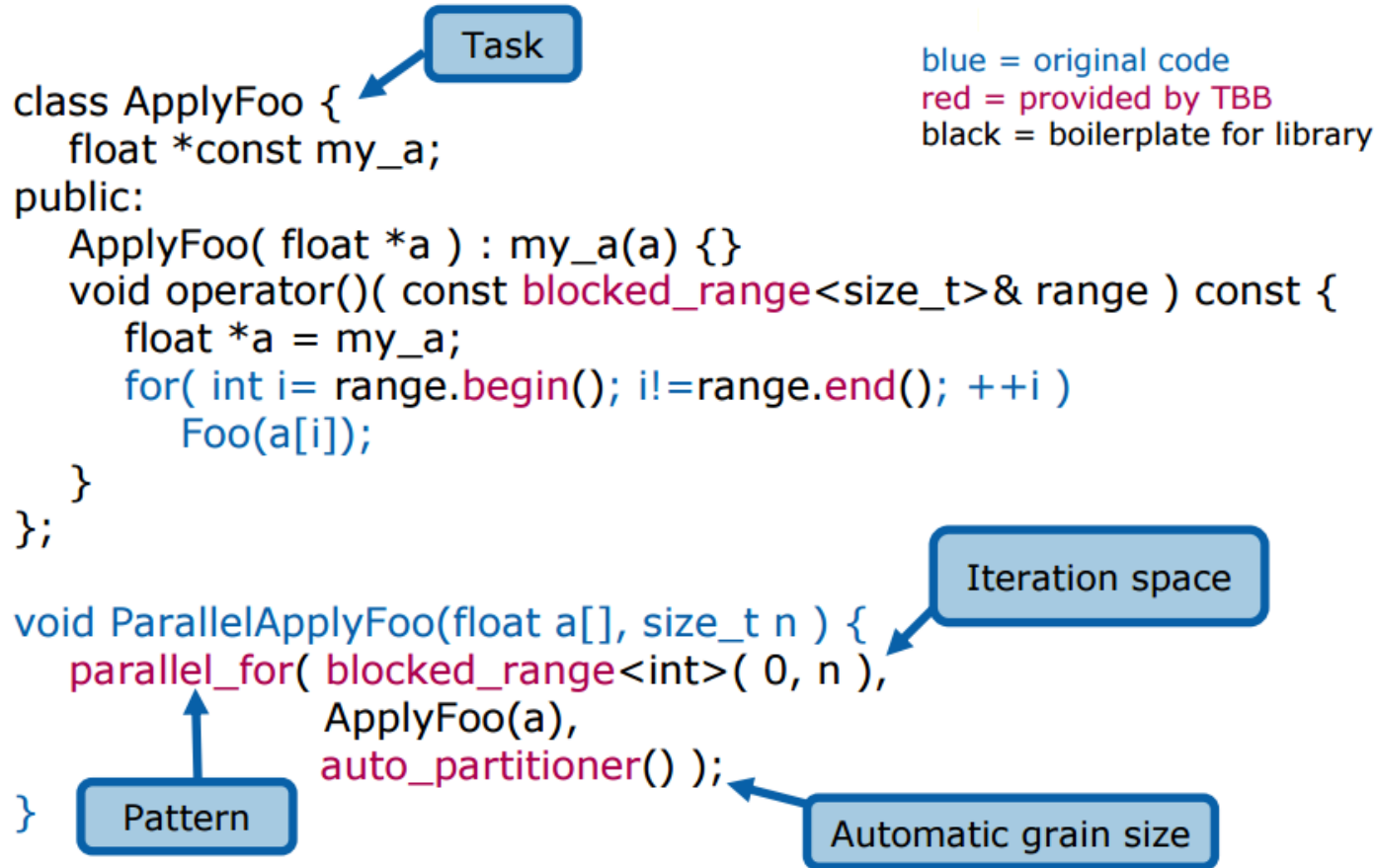
atomic, spin_mutex, spin_rw_mutex,
queuing_mutex, queuing_rw_mutex, mutex

Memory Allocation

cache_aligned_allocator
scalable_allocator

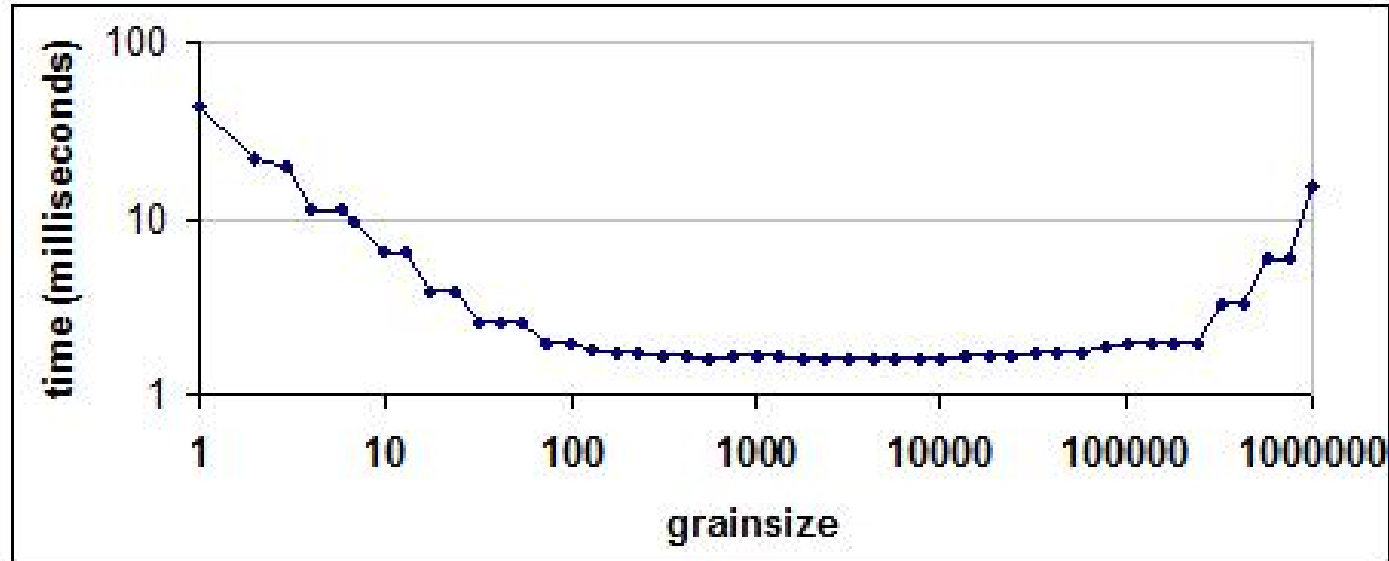
3. Patterns

parallel_for



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


- Automatic load balancing.
- Friendly cache usage.
- Adding more processors improve performance for existing code without recompilation.
- Range is generic, you can create your own ! Library gives you already implemented `blocked_range` and `blocked_range2D`.

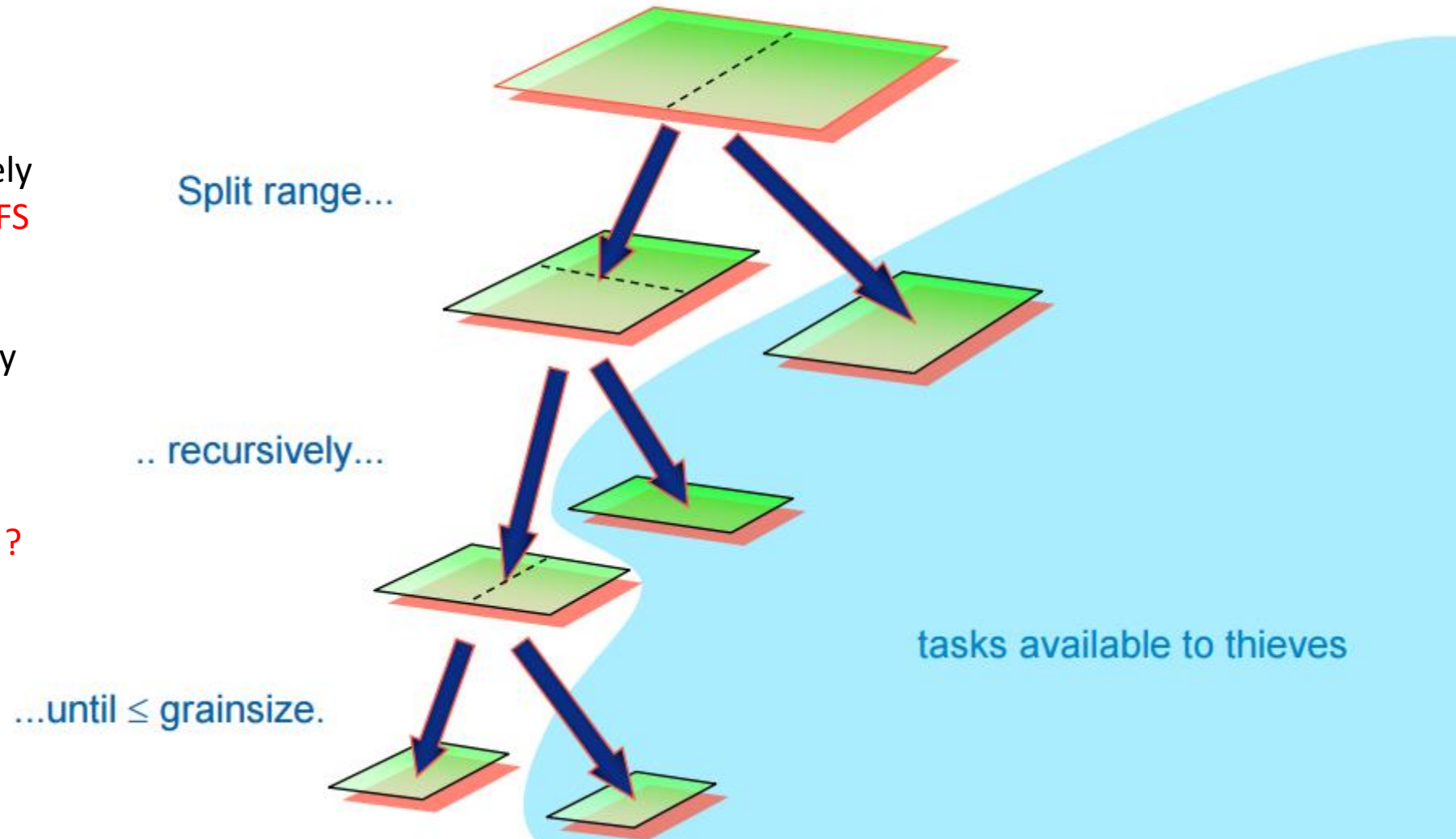


How this works on `blocked_range2d`

- Each worker keeps a deque of tasks
- Going down the tree recursively will generate new tasks in a **DFS order**.
- When a process is idle it will try to steal from others queue.

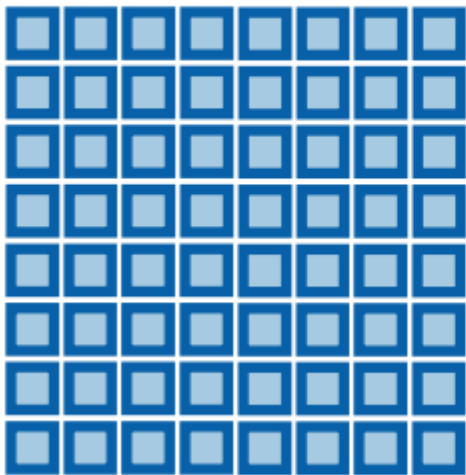
(later this course)

- Why a deque and not a queue ?
- What tasks are best to steal to optimize cache usage ?

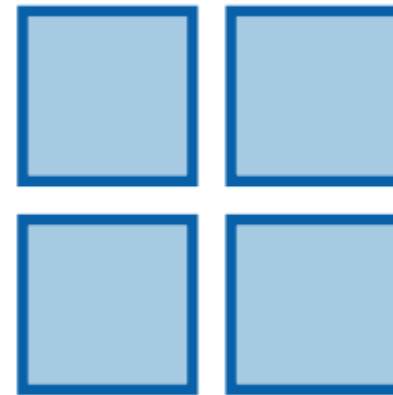


- Partition helps you split data in chunks and amortize overhead
- TBB offers two partitioners:
 - `auto_partitioner` : heuristically picks the grain size
`parallel_for(blocked_range(1, N), Body() , auto_partitioner ());`
 - `simple_partitioner` takes a manual grain size
`parallel_for(blocked_range(1, N, grain_size), Body());`

too fine ⇒
scheduling overhead dominates



too coarse ⇒
lose potential parallelism

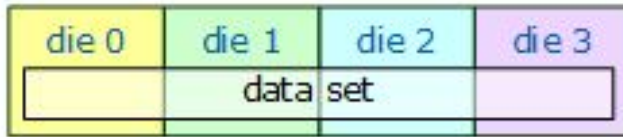


- Tuning is always needed. Start with single worker then project a graph with overhead per number of workers.

Memory bandwidth and cache affinity

Can significantly improve performance when:

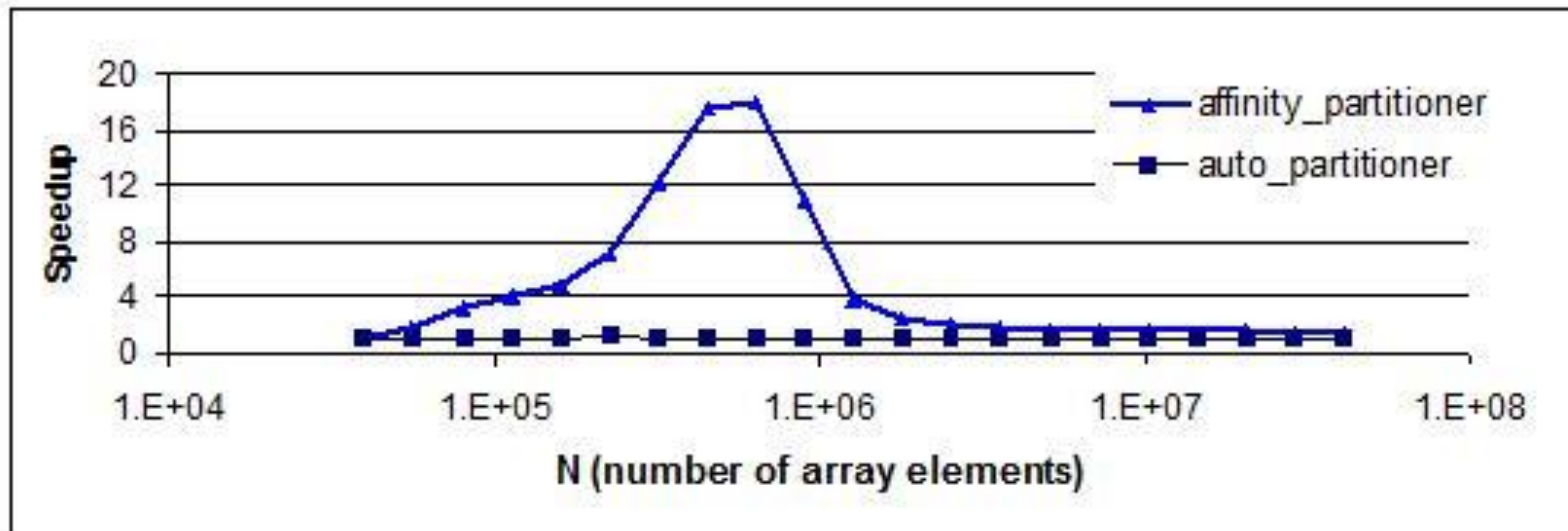
- Each computation does a few operations per data access and the data acted upon fits in cache.



- The loop is re-executed over the same data. (e.g. Game of Life example).

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

```
void TimeStepFoo( float a[], size_t n, int steps ) {  
    for( int t=0; t<steps; ++t )  
        ParallelApplyFoo( a, n );  
}
```



- $A[i] += B[i]$ for i in the range $[0, N)$
- If N small \Rightarrow parallel scheduling overhead dominates \Rightarrow no speedup
- If N big \Rightarrow data doesn't fit in cache memory
- Peak is in the middle ! Use it when there is **low ratio of computations vs memory accesses**

parallel_reduce

```
class MinIndexBody {
    const float *const my_a;
public:
    float value_of_min;
    long index_of_min;
    void operator()( const blocked_range<size_t>& r ) {
        const float* a = my_a;
        int end = r.end();
        for( size_t i=r.begin(); i!=end; ++i ) {
            float value = a[i];
            if( value<value_of_min ) {
                value_of_min = value;
                index_of_min = i;
            }
        }
    }
    MinIndexBody( MinIndexBody& x, split ) :
        my_a(x.my_a),
        value_of_min(FLT_MAX),
        index_of_min(-1)
    {}
    void join( const MinIndexBody& y ) {
        if( y.value_of_min<x.value_of_min ) {
            value_of_min = y.value_of_min;
            index_of_min = y.index_of_min;
        }
    }
    ...
};
```

accumulate result

split

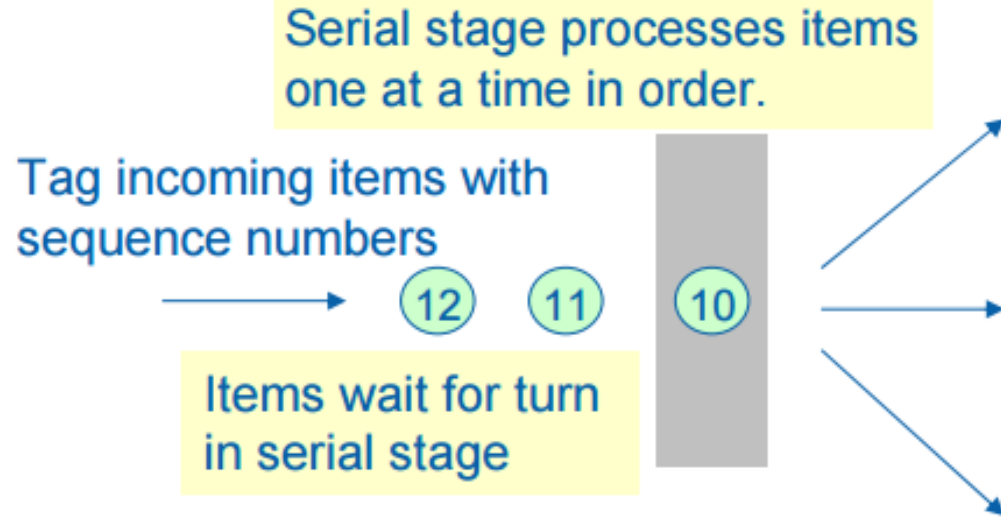
join

```
// Find index of smallest element in a[0...n-1] long ParallelMinIndex ( const float a[], size_t n )
{
    MinIndexBody mib(a);
    parallel_reduce(blocked_range(0,n,GrainSize), mib );
    return mib.index_of_min;
}
```

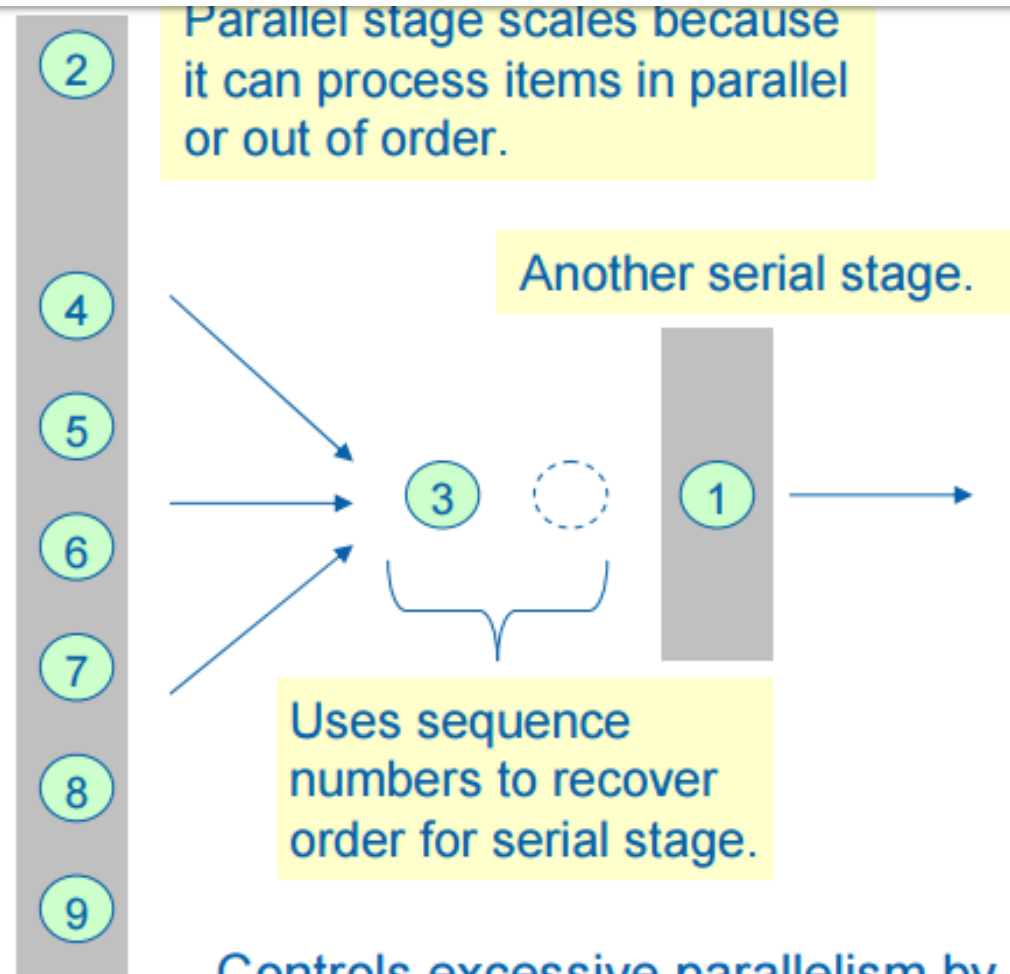
TBB also provides:

- parallel_scan
 - basic scan algorithm
- parallel_sort
 - Uses a quicksort since is cache friendlier than merge sort
- parallel_do
 - Used when you don't know loop bounds
 - Or tasks inside body can create new tasks (scalability is maintained !)

Parallel pipeline



Throughput limited by throughput of slowest serial stage.

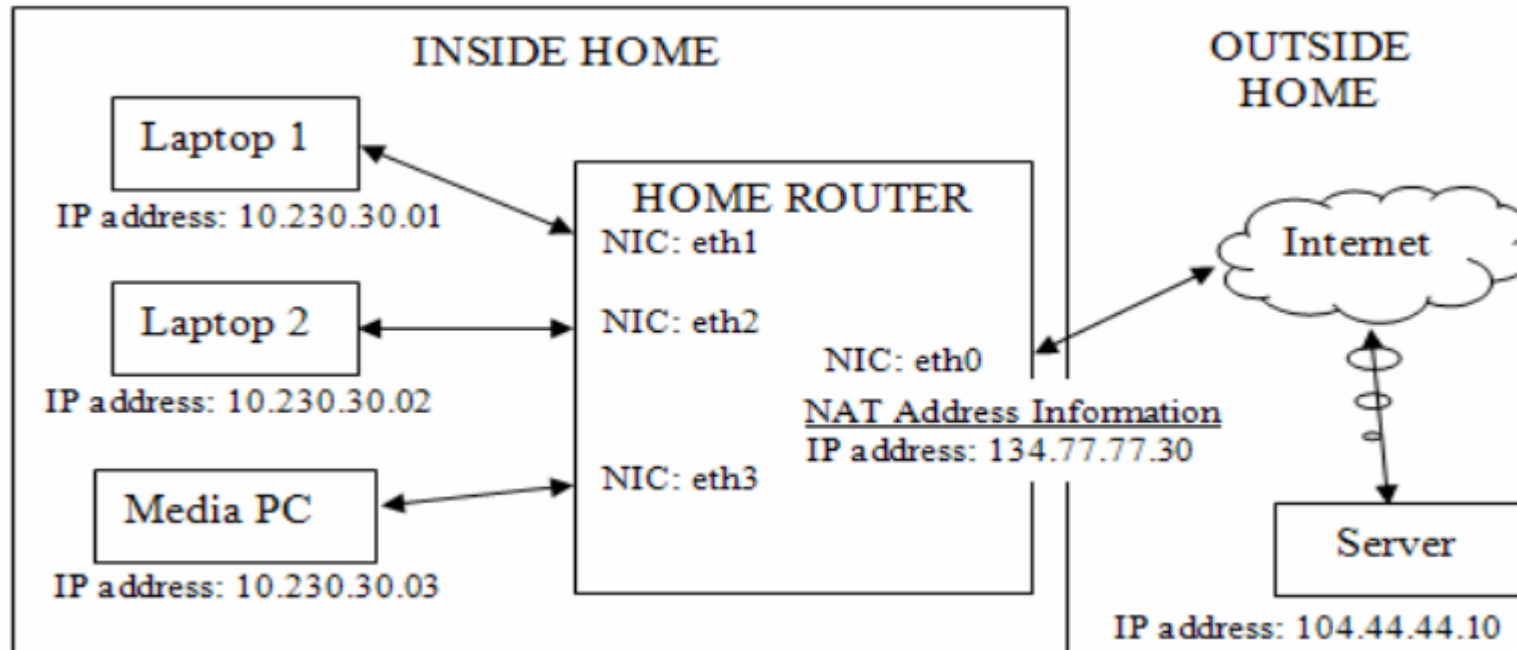


Controls excessive parallelism by limiting total number of items flowing through pipeline.

Pipeline for a Network Router



(Network address translation, Application Level gateway, Forwarding)



- Define the code for a pipeline stage
- Code reading packets from a streaming file
- Base constructor allows you to specify if the filter is serial or not

```
class get_next_packet : public tbb::filter {
    istream& in_file;
public:
    get_next_packet (ifstream& file) : in_file (file), filter (/*is_serial?*/ true) {}
    void* operator() (void*) {
        packet_trace* packet = new packet_trace ();
        in_file >> *packet; // Read next packet from trace file
        if (packet->packetNic == empty) { // If no more packets
            delete packet;
            return NULL;
        }
        return packet; // This pointer will be passed to the next stage
    }
};
```

- Router pipeline overview
- Run function allows you to specify the number of levels to run

```
#include "tbb/pipeline.h"
#include "router_stages.h"

void run_router (void) {
    tbb::pipeline pipeline; // Create TBB pipeline

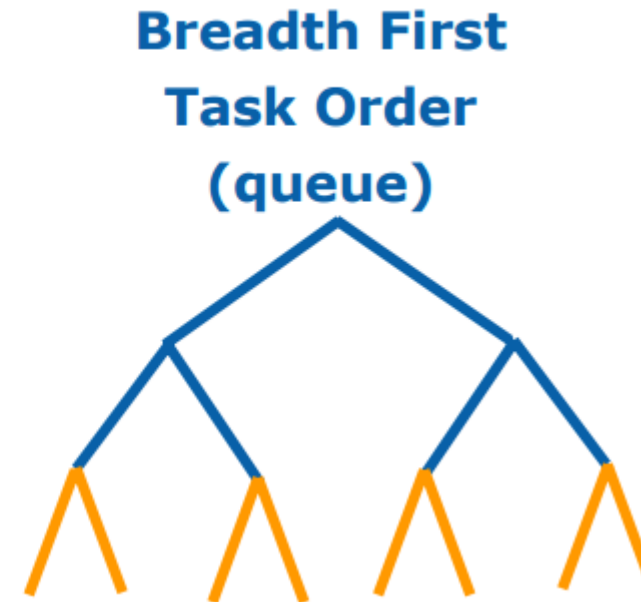
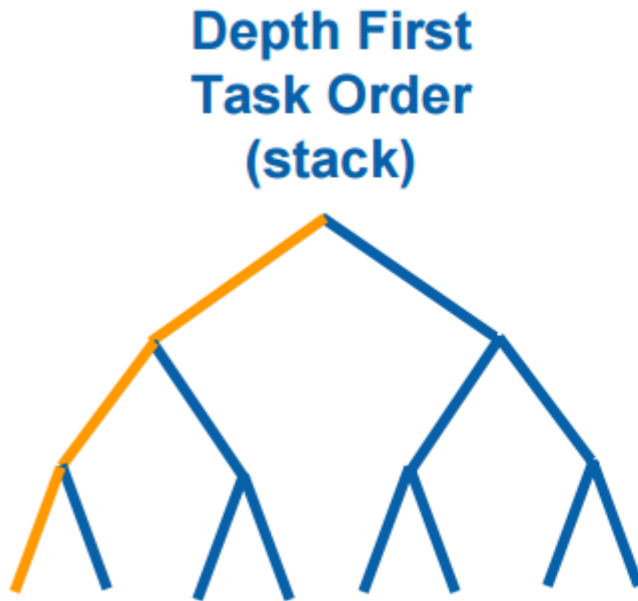
    get_next_packet receive_packet (in_file); // Create input stage
    pipeline.add_filter (receive_packet);      // Add input stage to pipeline

    translator network_address_translator (router_ip, router_nic, mapped_ports); // Create NAT stage
    pipeline.add_filter (network_address_translator); // Add NAT stage

    ... Create and add other stages to pipeline: ALG, FWD, Send ...

    pipeline.run (number_of_live_items); // Run Router
    pipeline.clear ();
}
```


3. Task execution and task stealing



- Compare cache locality and memory space needed

Depth First Task Order (stack)



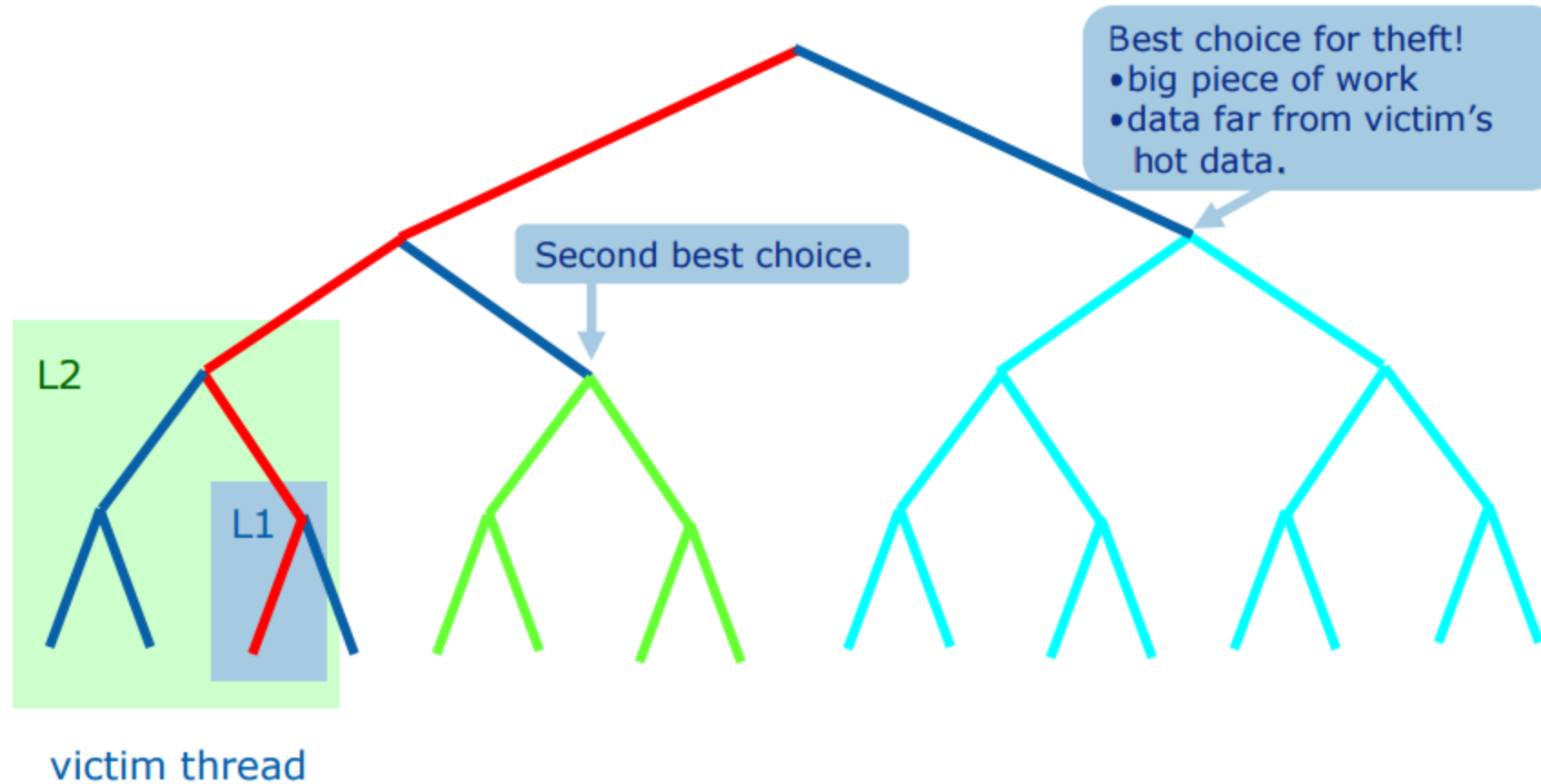
Small space
Excelent cache locality

Breadth First Task Order (queue)



Large space
Poor cache locality

Work Depth First; Steal Breadth First



- Each worker has a deque of tasks
- Steal from right end, work from left end
- Works well with nested parallelism

```

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

```

```

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

class FibTask: public Task {
public:
    const long n;
    long* const sum;
    FibTask( long n_, long* sum_ ) :
        n(n_), sum(sum_)
    {}
    Task* execute() { // Overrides virtual function Task::execute
        if( n<CutOff ) {
            *sum = SerialFib(n);
        } else {
            long x, y;
            FibTask& a = *new( allocate_child() ) FibTask(n-1,&x);
            FibTask& b = *new( allocate_child() ) FibTask(n-2,&y);
            set_ref_count(3); // 3 = 2 children + 1 for wait
            spawn( b );
            spawn_and_wait_for_all( a );
            *sum = x+y;
        }
        return NULL;
    }
};

```

5. Optimizations:

A) Recursive chain reaction

- If a master task tries to create N children directly it takes $O(N)$
- With a binary tree structure it takes $O(\log N)$
- Even if domains are not obviously tree structured, you can easily map them to trees.
- Check the example of `parallel_for`

B) Memory pools for tasks (check previous slide and overloaded new allocator)

C) Continuation passing

- Consider that a worker finished left branch and waits for the right branch to be finished by a worker....
- Idea: Why not creating a continuation task that could be added to the deque as a normal task ?

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

```
struct FibTask: public task {
    const long n;
    long* const sum;
    FibTask( long n_, long* sum_ ) :
        n(n_), sum(sum_)
    {}
    task* execute() {
        if( n<CutOff ) {
            *sum = SerialFib(n);
            return NULL;
        } else {
            // long x, y; This line removed
            FibContinuation& c =
                *new( allocate_continuation() ) FibContinuation(sum);
            FibTask& a = *new( c.allocate_child() ) FibTask(n-2,&c.x);
            FibTask& b = *new( c.allocate_child() ) FibTask(n-1,&c.y);
            // Set ref_count to "two children plus one for the wait".
            c.set_ref_count(2);
            spawn( b );
            spawn( a );
            // *sum = x+y; This line removed
            return NULL;
        }
    }
};
```

D) Scheduler bypass

- Specify directly the next task to run by returning it
- In the previous example after b is spawned:
 - a) Push task “a” on deque
 - b) Return from method execute()
 - c) Pop task “a” and execute it (unless stolen)
- Drawbacks:
 - Useless overhead for push/pop “a”
 - Hurt locality by allowing stealing **without** adding significant parallelism.
- Solution:
 - Spawn only “b”
 - Leave continuation as it was
 - Return “a” => after “b” is finished “a” will be executed then continuation.

```
struct FibTask: public task {
    ...
    task* execute() {
        if( n<CutOff ) {
            *sum = SerialFib(n);
            return NULL;
        } else {
            FibContinuation& c =
                *new( allocate_continuation() ) FibContinuation(sum);

            FibTask& a = *new( c.allocate_child() ) FibTask(n-2,&c.x);
            FibTask& b = *new( c.allocate_child() ) FibTask(n-1,&c.y);
            // Set ref_count to "two children".
            c.set_ref_count(2);
            spawn( b );
            // spawn( a ); This line removed
            // return NULL; This line removed
            return &a;
        }
    }
};
```

E) Recycling

- After creating the continuation task “c”, execute does these four steps:
 - a) Create task “a”
 - b) Create and spawn child task “b”
 - c) Return from execute() with pointer to task “a”
 - d) Destroy parent task
- But this allocate/deallocate will create memory fragmentation => limited performance
- Solution:
 - Recycle the parent as “a” (use “this” to create “a”)

```
struct FibTask: public task {
    long n;
    long* sum;
    ...
    task* execute() {
        if( n < CutOff ) {
            *sum = SerialFib(n);
            return NULL;
        } else {
            FibContinuation& c =
                *new( allocate_continuation() ) FibContinuation(sum);
            // FibTask& a = *new( c.allocate_child() ) FibTask(n-2,&c.x);

            FibTask& b = *new( c.allocate_child() ) FibTask(n-1,&c.y);
            recycle_as_child_of(c);
            n -= 2;
            sum = &c.x;
            // Set ref_count to "two children".
            c.set_ref_count(2);
            spawn( b );
            // return &a; This line removed
            return this;
        }
    }
};
```


F) Empty Tasks

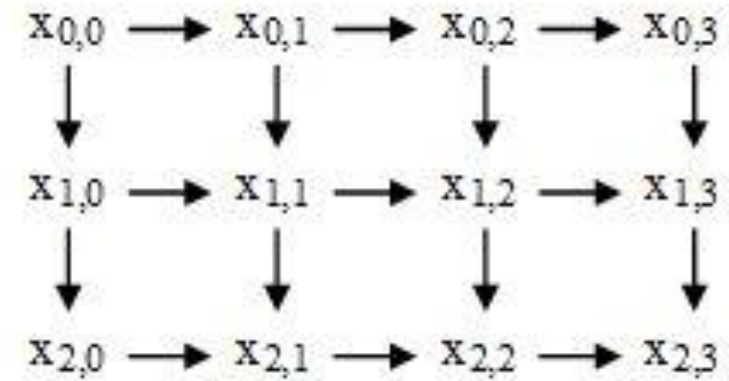
- Used for synchronization
- `parallel_for` uses it to create two children then uses an empty task as continuation

```
// Task that does nothing. Useful for synchronization.  
class empty_task: public task {  
    /*override*/ task* execute() {  
        return NULL;  
    }  
};
```

6. Acyclic Graphs of Tasks

- Can be used to schedule any kind of acyclic graphs:
- The key is to use the reference counting

```
class DagTask: public tbb::task {
public:
    const int i, j;
    // input[0] = sum from above, input[1] = sum from left
    double input[2];
    double sum;
    // successor[0] = successor below, successor[1] = successor to right
    DagTask* successor[2];
    DagTask( int i_, int j_ ) : i(i_), j(j_) {
        input[0] = input[1] = 0;
    }
    task* execute() {
        __TBB_ASSERT( ref_count()==0, NULL );
        sum = i==0 && j==0 ? 1 : input[0]+input[1];
        for( int k=0; k<2; ++k )
            if( DagTask* t = successor[k] ) {
                t->input[k] = sum;
                if( t->decrement_ref_count()==0 )
                    spawn( *t );
            }
        return NULL;
    }
};
```



```
double BuildAndEvaluateDAG() {
    DagTask* x[M][N];
    for( int i=M; --i>=0; )
        for( int j=N; --j>=0; ) {
            x[i][j] = new( tbb::task::allocate_root() ) DagTask(i,j);
            x[i][j]->successor[0] = i+1<M ? x[i+1][j] : NULL;
            x[i][j]->successor[1] = j+1<N ? x[i][j+1] : NULL;
            x[i][j]->set_ref_count((i>0)+(j>0));
        }
    // Add extra reference to last task, because it is waited on
    // by spawn_and_wait_for_all.
    x[M-1][N-1]->increment_ref_count();
    // Wait for all but last task to complete.
    x[M-1][N-1]->spawn_and_wait_for_all(*x[0][0]);
    // Last task is not executed implicitly, so execute it explicitly.
    x[M-1][N-1]->execute();
    double result = x[M-1][N-1]->sum;
    // Destroy last task.
    task::destroy(*x[M-1][N-1]);
    return result;
}
```

- General data flow and dependency:

https://www.threadingbuildingblocks.org/docs/help/tbb_userguide/Flow_Graph.html

- Allows functional parallelism on nodes
- Supports buffering, messages and many other nice stuff

7. Synchronization

- Mutex: **spin_mutex** / **queuing_mutex** (unfair vs fair)
- Multiple reader and single writer : **spin_rw_mutex**, **queuing_rw_mutex**
- Use **atomic<T>** whenever possible

```
struct Foo {  
    atomic<int> refcount;  
};  
void RemoveRef( Foo& p ) {  
    --p. refcount;  
    if( p. refcount ==0 ) delete &p;  
}  
void RemoveRef(Foo& p ) {  
    if( --p. refcount ==0 ) delete &p;  
}
```

Wrong !



8. Concurrent containers

- Remember: STL containers are not safe under concurrent operations
- TBB provides fine-grained locking and lockless operations where possible
- STL interfaces are inherently not concurrency -friendly

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

tbb::concurrent_queue has pop_if_present

- TBB has concurrent_vector, concurrent_array, concurrent_hash (map)
- Concurrent reads are allowed for maximum performance

Example: map strings to integers

// Define hashing and comparison operations for the user type.

```
struct MyHashCompare {  
    static long hash( const char* x ) {  
        long h = 0;  
        for( const char* s = x; *s; s++ )  
            h = (h*157)^*s;  
        return h;  
    }  
};
```

```
    static bool equal( const char* x, const char* y ) {  
        return strcmp(x,y)==0;  
    }  
};
```

```
typedef concurrent_hash_map<const char*,int,MyHashCompare> StringTable;
```

```
StringTable MyTable;
```

```
void MyUpdateCount( const char* x ) {  
    StringTable::accessor a;  
    MyTable.insert( a, x );  
    a->second += 1;  
}
```

**Multiple threads can
insert and update entries
concurrently.**

accessor object acts as a
smart pointer and a writer
lock: no need for explicit
locking.