Threading Building Blocks

A short intro to task-based parallel run-time systems

Outline

1. Introduction
2. Programming model
3. Internals

Outline

1. **Introduction**
2. Programming model
3. Internals

Challenges of parallel programming

Finding parallelism

- manually
- automatically

Expressing parallelism

- low-level (Pthreads, MPI)
- high-level (parallel RTSs)

Mapping parallelism

- creating, scheduling, terminating, etc., parallel tasks
- OS, RTS

Synchronization

- easy, as coarse-grain
- effective, as fine-grain
- deadlock-free (+convoying,priority inversion,etc.)
- composable

Requirements

- scalability
- high productivity
- correctness
- architectural awareness

Challenges of parallel programming

Finding parallelism

- manually
- automatically

Expressing parallelism

- low-level (Pthreads, MPI)
- high-level (parallel RTSs)

Mapping parallelism

- creating, scheduling, terminating, etc., parallel tasks
- OS, RTS

Synchronization

- easy, as coarse-grain
- effective, as fine-grain
- deadlock-free (+convoying,priority inversion,etc.)
- composable

Requirements

- scalability
- high productivity
- correctness
- architectural awareness

TBBs

Background

C++ template library for shared-memory parallel programming

- developed by Intel since 2004 (open-source since 2007)
- not a new language or extension
- portable on most C++ compilers, OSs and architectures

TBBs

Key concepts, I

Programmer defines ***tasks***, not threads

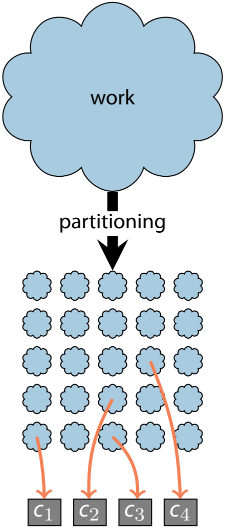
- focuses on ***describing*** parallelism
- RTS responsible for ***implementing*** parallelism
  - decomposes work to smaller tasks
  - schedules tasks to processors
  - sychronization
  - load balancing
  - system resources management (processors, memory)

TBBs

Key concepts, II

Designed for ***scalability***

- original work decomposed to C chunks, $C \gg N_{procs}$ (“*parallel slack*”)
- ensures that work will be always available for each processor added
- load balancing guarantees scalable performance



TBBs

Key concepts, III

Exploits the power and flexibility of *generic programming*

- provides templated, ready-to-use parallel algorithmic skeletons and structures
  - much like STL does for serial programs
- allows C++ Lambdas for enhanced ease-of-use
- does not require special compiler support

Outline

TBB 4.0 components

Generic parallel algorithms

parallel_for
parallel_reduce
parallel_scan
parallel_do
pipeline, parallel_pipeline
parallel_sort
parallel_invoke

Raw tasking

task
task_group
task_list

Flow graphs

graph
functional nodes
buffering nodes
split/join nodes

Concurrent containers

concurrent_unordered_map
concurrent_unordered_set
concurrent_has_map
concurrent_queue
concurrent_bounded_queue
concurrent_priority_queue
concurrent_vector

Synchronization primitives

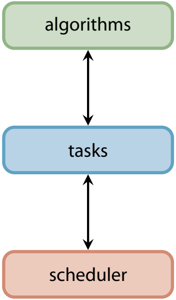
atomic
mutex
recursive_mutex
spin_mutex, spin_rw_mutex
queueing_mutex, queueing_rw_mutex

Memory allocation

tbb_allocator
cache_aligned_allocator
scalable_allocator

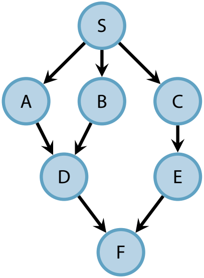
Tasks

- An elementary, independent item of work
- Much more lightweight than native threads
  - user-level, small-sized, non-preemptible, short-lived
- Basic bulding block of TBBs algorithms
- TBBs allow direct use of the low-level tasking API
  - creation of arbitrarily complex task graphs
- Two basic operations: `spawn` and `wait`



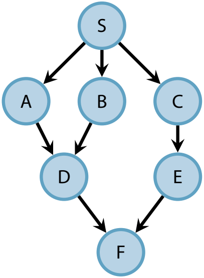
Tasks example

Generic task graph



Tasks example

Generic task graph




```
S();
task_group g;
g.run( [&]{ C(); E(); } );
g.run( [&]{
            task_group g1;
            g1.run( [&]{A();} );
            g1.run( [&]{B();} );
            g1.wait();
            D();
    });
g.wait();
F();
```

Tasks example

Recursive parallelism

```
long ParallelFib(long n) {
  if ( n < cutOff ) return SerialFib(n);
  else {
    int x,y;
    task_group g;
    g.run( [&]{ x = ParallelFib(n-1); });
    g.run( [&]{ y = ParallelFib(n-2); });
    g.wait();
    return x+y;
  }
}
```

Tasks example

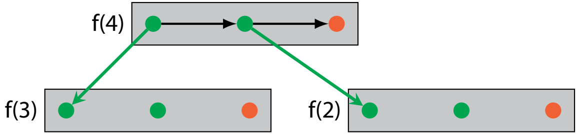
Recursive parallelism

```
long ParallelFib(long n) {
  if ( n < cutOff ) return SerialFib(n);
  else {
    int x,y;
    task_group g;
    g.run( [&]{ x = ParallelFib(n-1); });
    g.run( [&]{ y = ParallelFib(n-2); });
    g.wait();
    return x+y;
  }
}
```

Tasks example

Recursive parallelism

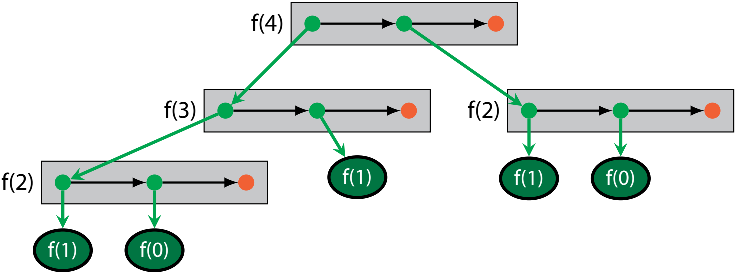
```
long ParallelFib(long n) {
  if ( n < cutOff ) return SerialFib(n);
  else {
    int x,y;
    task_group g;
    g.run( [&]{ x = ParallelFib(n-1); });
    g.run( [&]{ y = ParallelFib(n-2); });
    g.wait();
    return x+y;
  }
}
```



Tasks example

Recursive parallelism

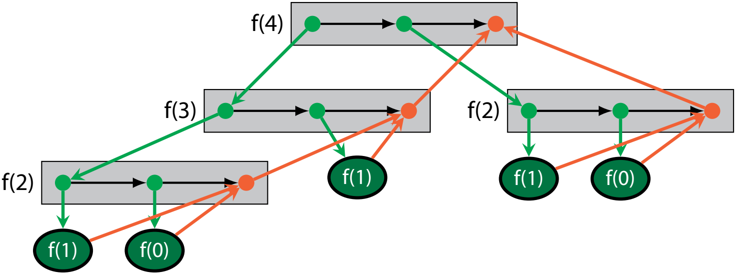
```
long ParallelFib(long n) {
  if ( n < cutOff ) return SerialFib(n);
  else {
    int x,y;
    task_group g;
    g.run( [&]{ x = ParallelFib(n-1); });
    g.run( [&]{ y = ParallelFib(n-2); });
    g.wait();
    return x+y;
  }
}
```



Tasks example

Recursive parallelism

```
long ParallelFib(long n) {
  if ( n < cutOff ) return SerialFib(n);
  else {
    int x,y;
    task_group g;
    g.run( [&]{ x = ParallelFib(n-1); });
    g.run( [&]{ y = ParallelFib(n-2); });
    g.wait();
    return x+y;
  }
}
```



Tasks example, low-level interface

Recursive parallelism

Root task creation

```
long n, sum;
FibTask& r = *new (allocate_root())
                      FibTask(n,&sum);
spawn_root_and_wait();
cout « sum;
```

Serial code

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

Class definition

```
class 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);
    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);
      spawn(b);
      spawn(a);
      wait_for_all();
      *sum = x+y;
    }
    return NULL;
  }
};
```

Quiz

Parent task

```
...
FibTask& a = *new(allocate_child()) FibTask(n-1,&x);
FibTask& b = *new(allocate_child()) FibTask(n-2,&y);
set_ref_count(3);
spawn(b);
spawn(a);
wait_for_all();
*sum = x+y;
...
```

Q How does execution continue after wait_for_all() is called?

- Where is the parent suspended?
- How does its worker continue to process other tasks?
- How does the parent get notified and resume after children completion?

Parallel algorithms

Loop parallelization

`parallel_for`

- divides iteration space into smaller chunks and executes them in parallel
- template function parametrized with:
  - iteration range object
  - work description object

```
template <typename Range, typename Body>
void parallel_for(const Range& R, const Body& B);
```

Parallel algorithms

Loop parallelization

`parallel_for`

- divides iteration space into smaller chunks and executes them in parallel
- template function parametrized with:
  - iteration range object
  - work description object

```
template <typename Range, typename Body>
void parallel_for(const Range& R, const Body& B);
```

requirements for Body B

```
B::B(const F&)
B::~B()
void B::operator()(Range& subrange) const
```

requirements for Range R

```
R(const R&)
R::~R()
bool R::empty() const
bool R::is_divisible() const
R::R(R& r, split)
```

Example

Loop parallelization

```
for ( size_t i = 0; i != n; ++i )
   a[i] = a[i] + b[i];
```

Example

Loop parallelization

```
for ( size_t i = 0; i != n; ++i )
   a[i] = a[i] + b[i];
```

⇓

```
parallel_for (
  blocked_range<size_t>(0,n),
  [=]( const blocked_range<size_t>& r) {
    for ( size_t i = r.begin(); i != r.end(); ++i )
      a[i] = a[i] + b[i];
  });
```

Parallel algorithms

Parallel reduction

```
template <typename Range, typename Value,
          typename Func, typename Reduction>
Value parallel_reduce(const Range& R, const Value& identity,
                      const Func& f, const Reduction& red );
```

Semantics

Value Identity	Identity element for Func::operator()
Value Func::operator()(const Range& range, const Value& x)	Accumulate result for subrange, starting with initial value x
Value Reduction::operator()(const Value& x, const Value& y);	Combine results x and y

Example

Parallel reduction

```
sum = 0.0;
for ( size_t i = 0; i != n; ++i )
   sum += a[i];
```

Example

Parallel reduction

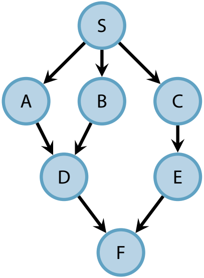
```
sum = 0.0;
for ( size_t i = 0; i != n; ++i )
   sum += a[i];
```

$$\Downarrow$$

```
parallel_reduce(
    blocked_range<float*>(a, a+n),
    0.0,
    [](const blocked_range<float*>& r, float init)->float {
        for ( float* v = r.begin(); v != r.end(); ++v )
           init += *v;
        return init;
    },
    [](float x, float y)->float { return x+y; }
    );
```

Flow graph

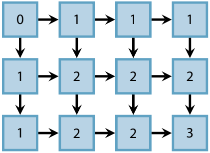
Generic task graph, revisited




```
graph g;
broadcast_node<continue_msg> s;

continue_node<continue_msg> a(g,A());
continue_node<continue_msg> b(g,B());
continue_node<continue_msg> c(g,C());
continue_node<continue_msg> d(g,D());
continue_node<continue_msg> e(g,E());
continue_node<continue_msg> f(g,F());
make_edge(s,a);
make_edge(s,b);
make_edge(s,c);
make_edge(a,d);
make_edge(b,d);
make_edge(c,e);
make_edge(d,f);
make_edge(e,f);

S();
s.try_put(continue_msg()); //fire!
g.wait_for_all();
```

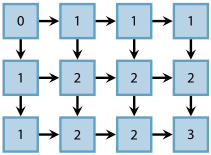
Task graph with fixed dependences ("wavefront")

task objects + reference counts



Task graph with fixed dependences ("wavefront")

task objects + reference counts



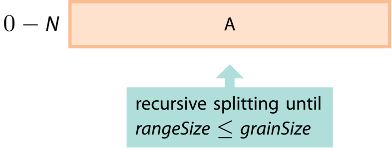

```
MeshTask* Mesh[3][4];
//for all tasks in Mesh:
// allocate
// initialize south,east pointers
// set reference counters

//wait for all but last task to complete
Mesh[2][3]->spawn_and_wait_for_all(
            *Mesh[0][0]);

//execute last task
Mesh[2][3]->execute();
```

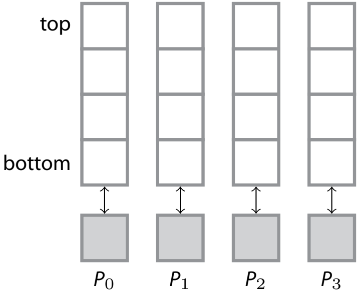
```
class MeshTask: public task {
  public:
    const int i,j; //coordinates
    MeshTask *south, *east;

  task* execute() {
    double north_val = (i==0) ? 0 : A[i-1][j];
    double west_val = (j==0) ? 0 : A[i][j-1];
    A[i][j] = do_work(north_val, west_val);

    if ( south != NULL )
      if (!south->decrement_ref_count())
        spawn(*south);
    if ( east != NULL )
      if (!east->decrement_ref_count())
        spawn(*east);

    return NULL;
  }
}
```

Outline

Parallel for

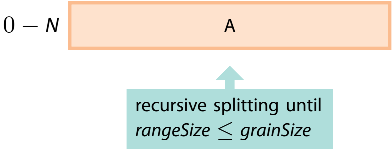
Recursive splitting

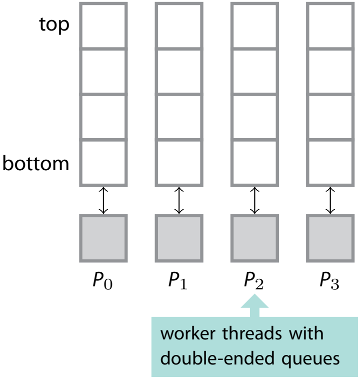




Parallel for

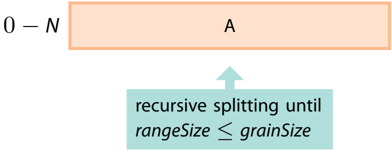
Recursive splitting

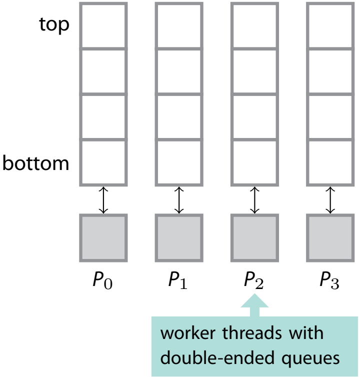




Parallel for

Recursive splitting

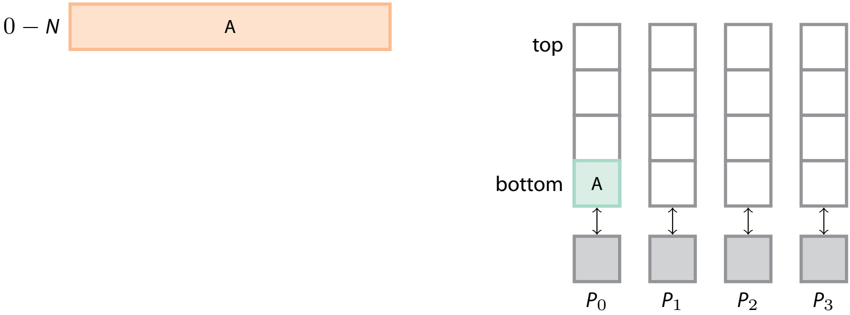




- at each recursion step, a range is split in 2 subranges
- new tasks placed at bottom
- each worker takes a task from its local queue and ***executes*** it
- if queue empty, it ***steals*** one from a random victim

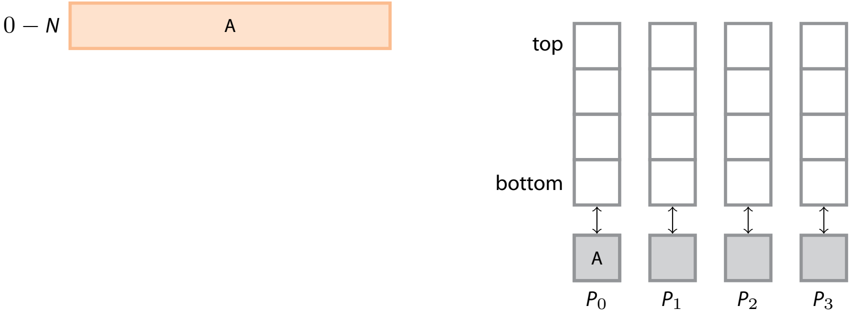
Parallel for

Recursive splitting



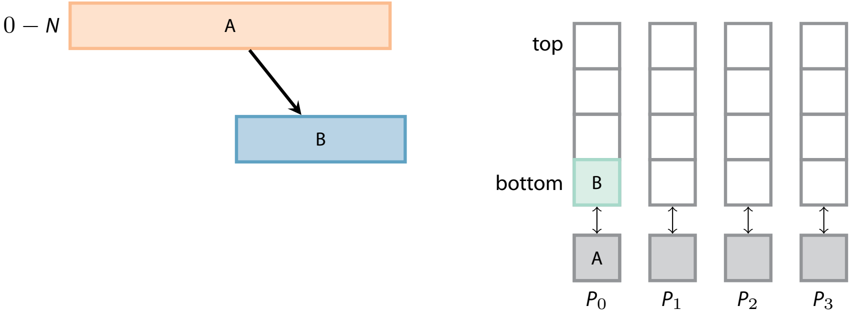
Parallel for

Recursive splitting



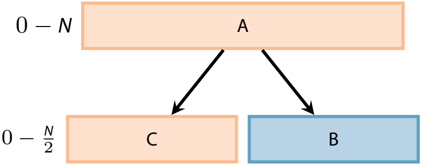
Parallel for

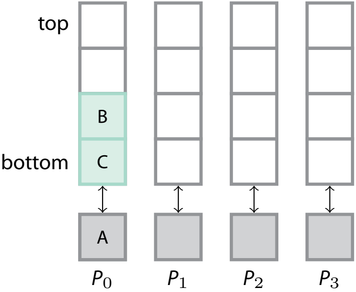
Recursive splitting



Parallel for

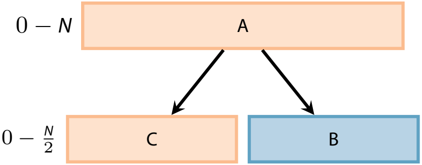
Recursive splitting

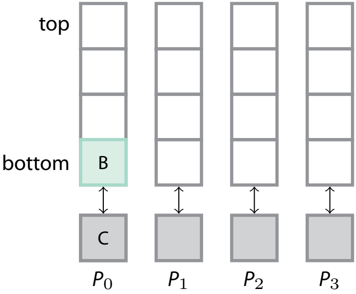




Parallel for

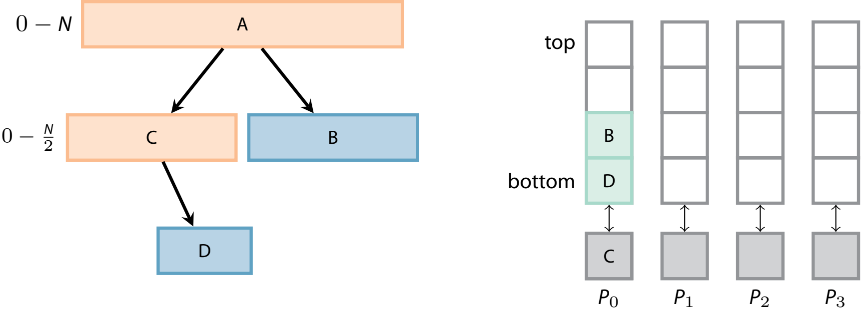
Recursive splitting





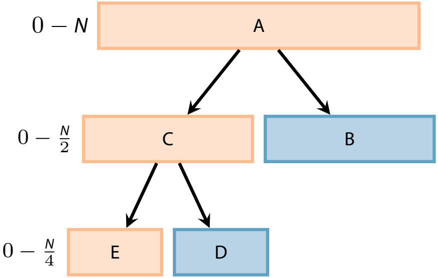
Parallel for

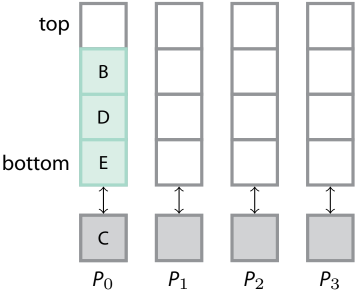
Recursive splitting



Parallel for

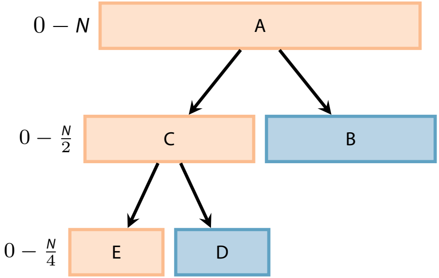
Recursive splitting

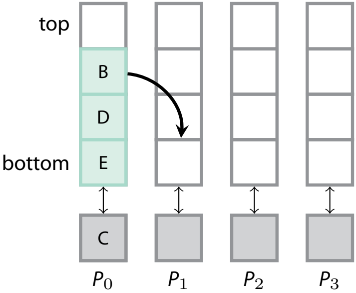




Parallel for

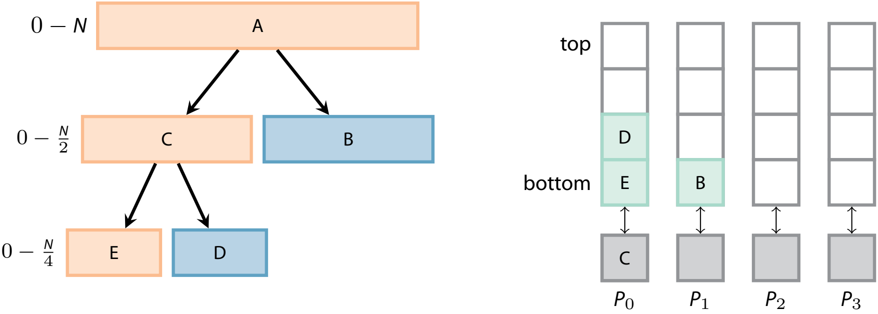
Recursive splitting





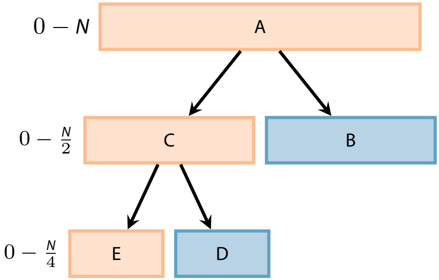
Parallel for

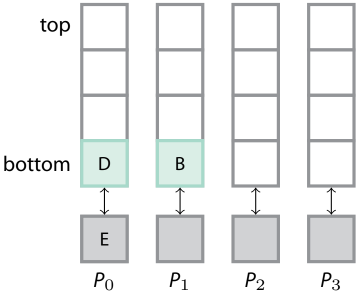
Recursive splitting



Parallel for

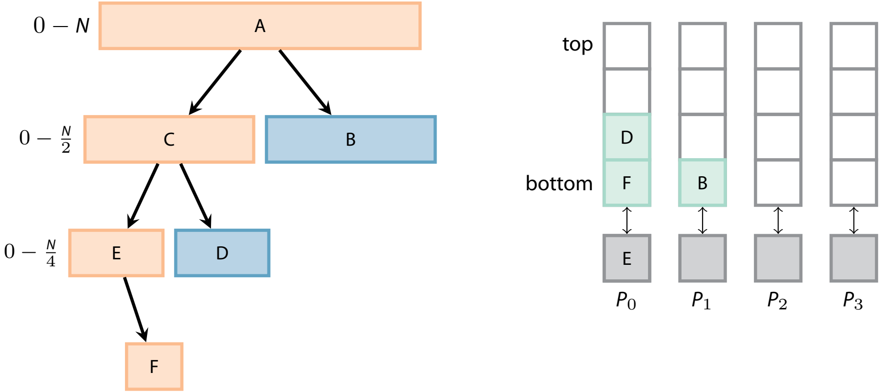
Recursive splitting





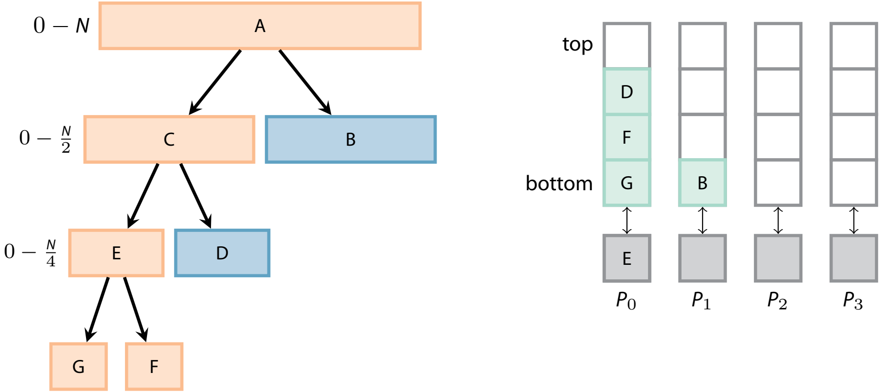
Parallel for

Recursive splitting



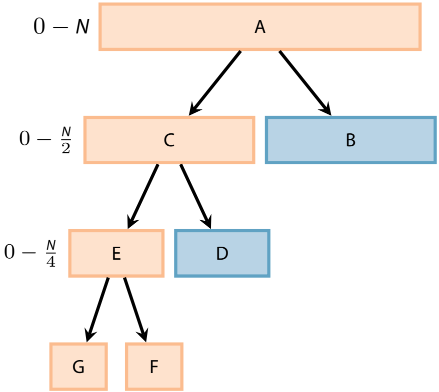
Parallel for

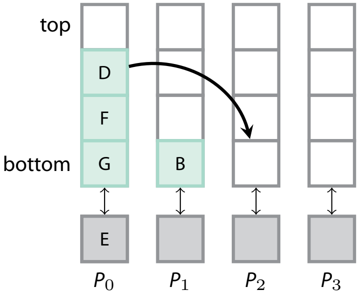
Recursive splitting



Parallel for

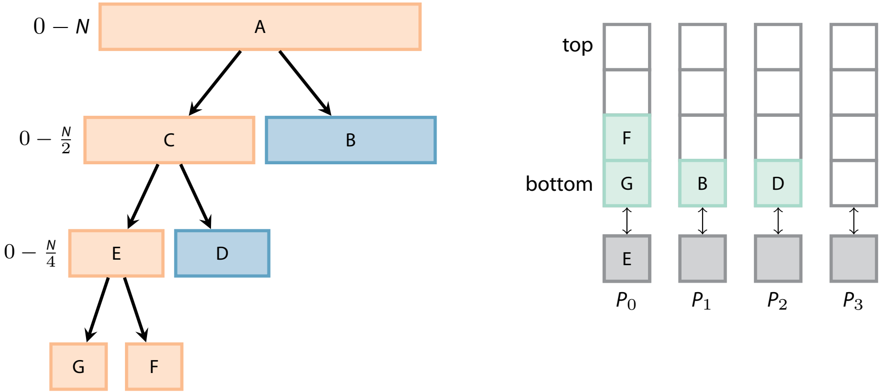
Recursive splitting





Parallel for

Recursive splitting



Key mechanisms

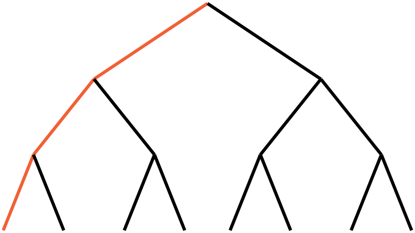
Work stealing

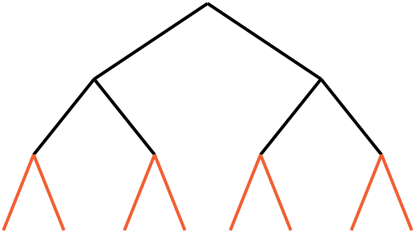
- guarantees load balancing

Recursive splitting

- allows processing arbitrarily small chunks
- enables optimal cache usage (“cache-oblivious algorithms”)

Possible task execution orders





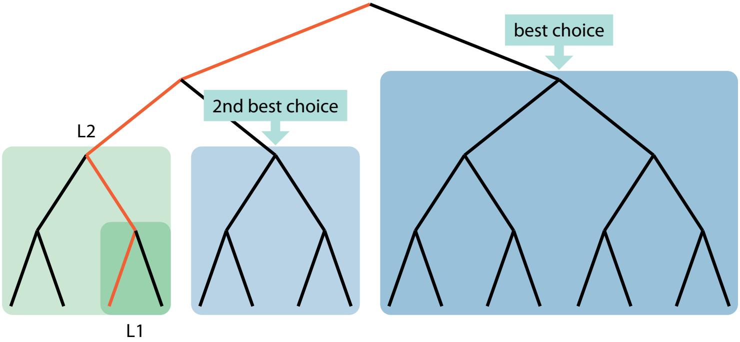
Depth-first

- small space
- excellent cache locality
- no parallelism

Breadth-first

- large space
- poor cache locality
- maximum parallelism

Work depth-first, steal breadth-first



Stealing from top guarantees:

- big piece of work $\Rightarrow$ more efficient load balancing
- data far from victim's hot data