How to Solve Complex Problems in Parallel (Parallel Divide and Conquer *and* Task Parallelism)

Kenjiro Taura

Contents

Contents

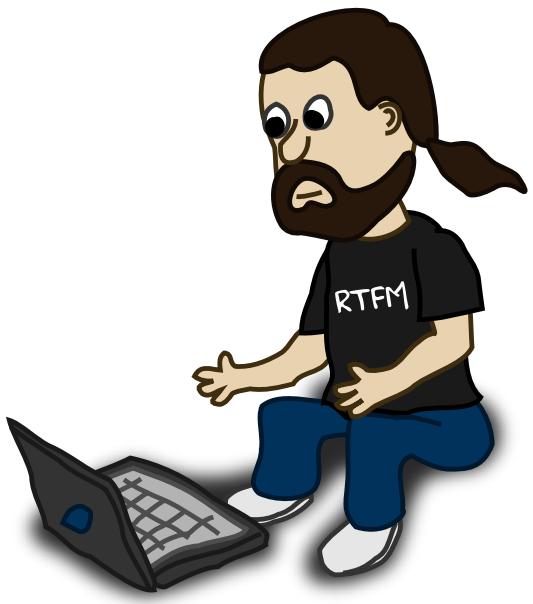
Goals

learn:

- the power of divide and conquer paradigm, combined with task parallelism, with concrete examples,
- how to write task parallel programs in Intel TBB,
- and how to reason about the speedup of task parallel programs
  - work
  - critical path length
  - Greedy Scheduler theorem

Hands-on exercise next week

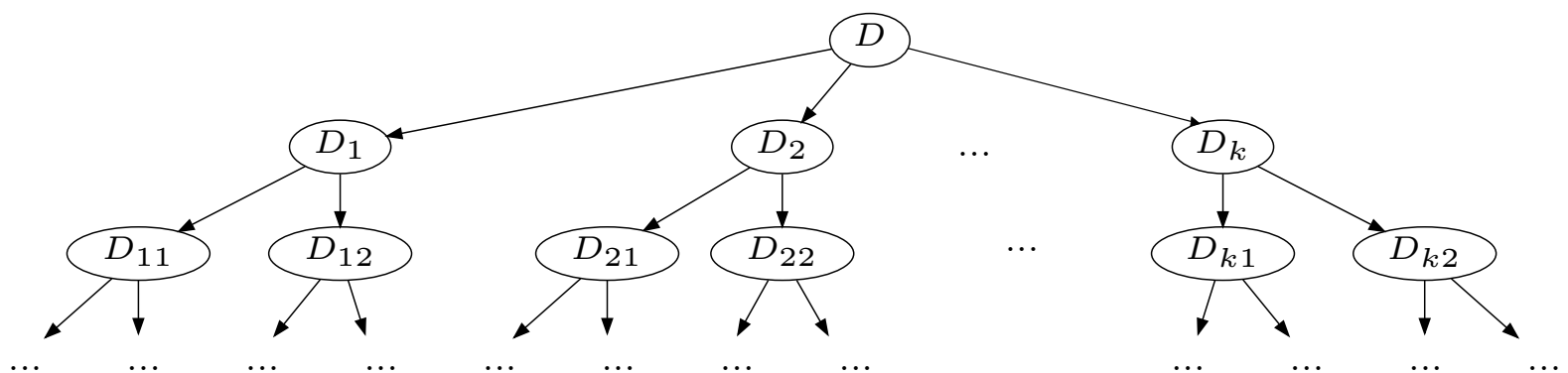
- please bring your laptop with SSH
- details on the necessary preparation are (hopefully) on the home page until the weekend



Divide and conquer algorithms

- "Divide and conquer" is the single most important design paradigm of algorithms

```
1  answer solve(D) {
2    if ( trivial (D)) {
3      return trivially_solve (D);
4    } else {
5      D_1, ..., D_k = divide(D); // divide the problem into sub problems
6      a_1 = solve(D_1); ...; a_k = solve(D_k); // solve them
7      return combine(a_1, ..., a_k); // combine sub answers
8    }
9  }
```



Benefits of "divide and conquer" thinking

Divide and conquer ...

- often helps you *come up with* an algorithm
- is easy to program, with *recursions*
- is often easy to *parallelize*, once you have a recursive formulation and a parallel programming language that support it (*task parallelism*)
- often has a good *locality* of reference, both in serial and parallel execution

Some examples

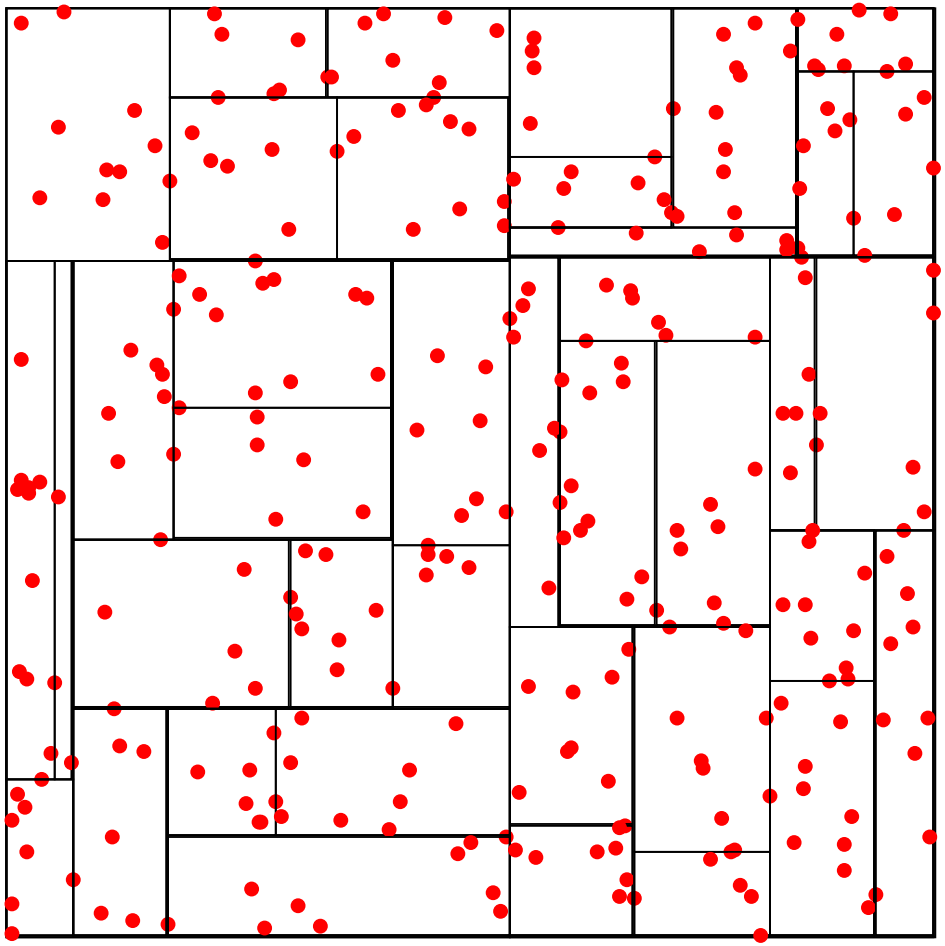
- quick sort, merge sort
- matrix multiply, LU factorization, eigenvalue
- FFT, polynomial multiply, big int multiply
- maximum segment sum, find median
- k-d tree
- ...

Contents

Contents

k-d tree

- A data structure to hierarchically organize points (to facilitate "nearest neighbor" searches) (usually in 2D or 3D space)
- Each node represents a rectangle region



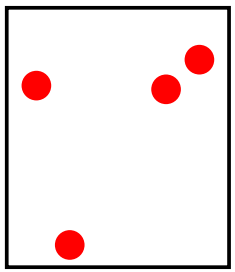
k-d tree construction

- Input:
  - P: an array of points (no particular order)
  - R: a bounding box of P
- Output:
  - t: a k-d tree for P
- Properties of k-d trees

k-d tree construction

- Input:
  - P: an array of points (no particular order)
  - R: a bounding box of P
- Output:
  - t: a k-d tree for P
- Properties of k-d trees
  - a leaf has $\leq c$ points

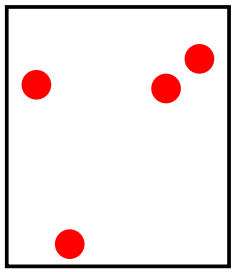
Leaf:



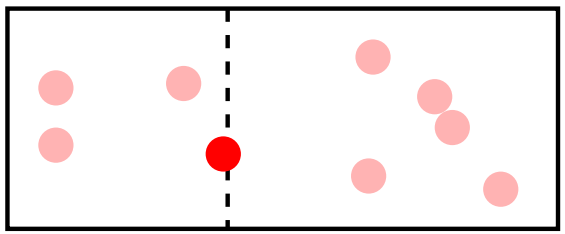
k-d tree construction

- Input:
  - P: an array of points (no particular order)
  - R: a bounding box of P
- Output:
  - t: a k-d tree for P
- Properties of k-d trees
  - a leaf has $\leq c$ points
  - an internal node has one point of its own plus one or two children
  - an internal node is split into two subspaces through its point

Leaf:



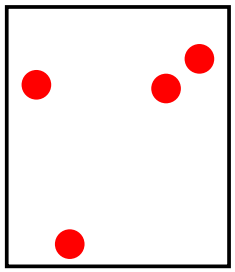
Internal:



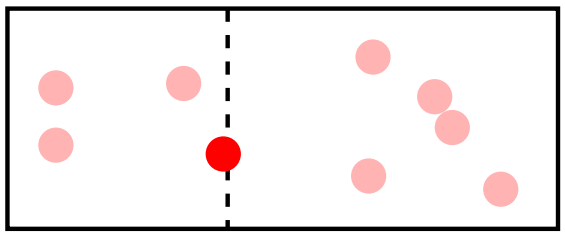
k-d tree construction

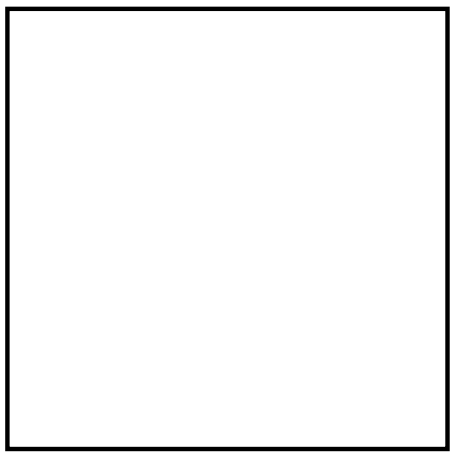
- Input:
  - P: an array of points (no particular order)
  - R: a bounding box of P
- Output:
  - t: a k-d tree for P
- Properties of k-d trees
  - a leaf has $\leq c$ points
  - an internal node has one point of its own plus one or two children
  - an internal node is split into two subspaces through its point
  - axis along which to split nodes are chosen cyclically (first along x-axis, then along y-axis, and so on)

Leaf:



Internal:

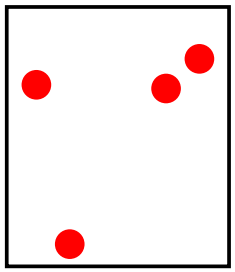




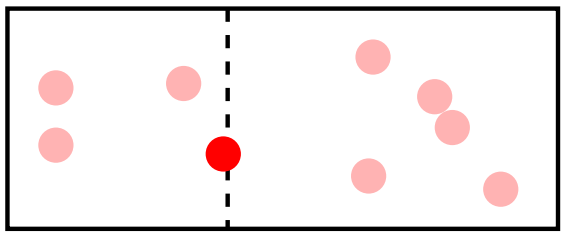
k-d tree construction

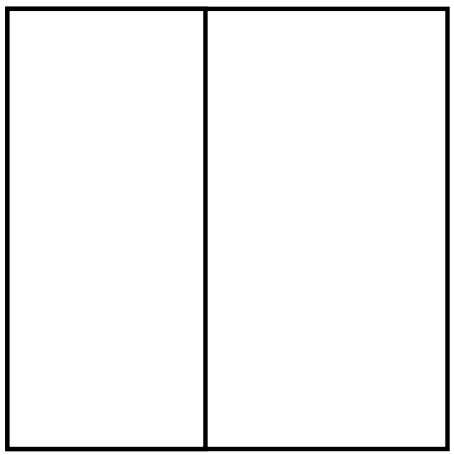
- Input:
  - P: an array of points (no particular order)
  - R: a bounding box of P
- Output:
  - t: a k-d tree for P
- Properties of k-d trees
  - a leaf has $\leq c$ points
  - an internal node has one point of its own plus one or two children
  - an internal node is split into two subspaces through its point
  - axis along which to split nodes are chosen cyclically (first along x-axis, then along y-axis, and so on)

Leaf:



Internal:

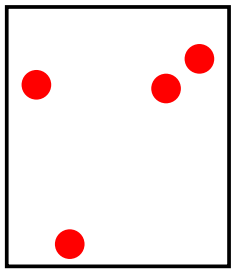




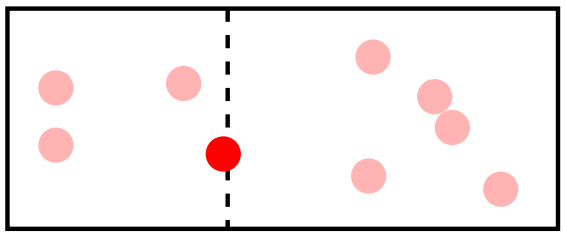
k-d tree construction

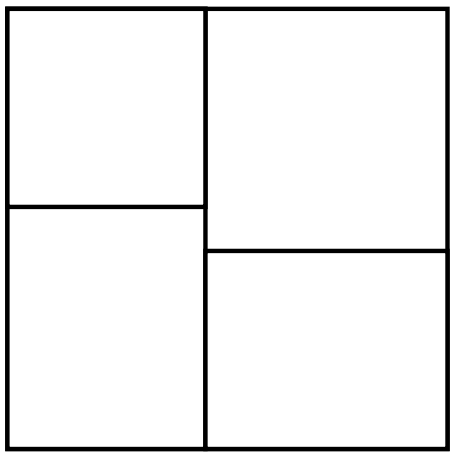
- Input:
  - P: an array of points (no particular order)
  - R: a bounding box of P
- Output:
  - t: a k-d tree for P
- Properties of k-d trees
  - a leaf has $\leq c$ points
  - an internal node has one point of its own plus one or two children
  - an internal node is split into two subspaces through its point
  - axis along which to split nodes are chosen cyclically (first along x-axis, then along y-axis, and so on)

Leaf:



Internal:

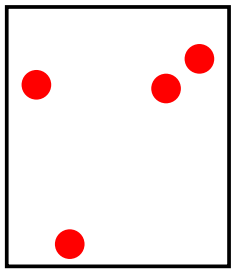




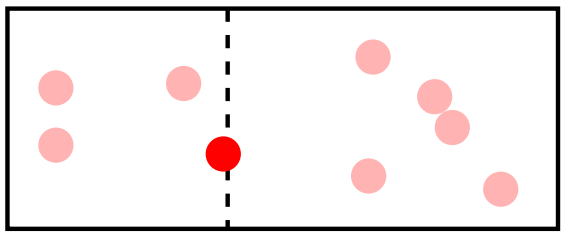
k-d tree construction

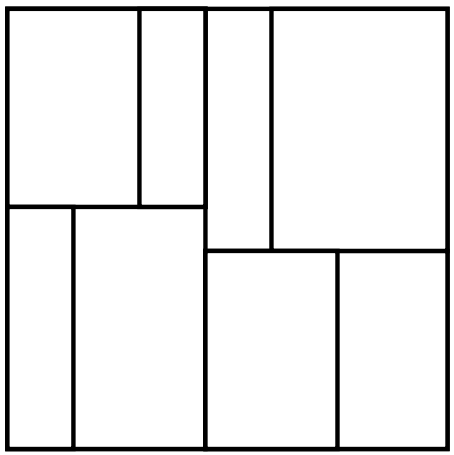
- Input:
  - P: an array of points (no particular order)
  - R: a bounding box of P
- Output:
  - t: a k-d tree for P
- Properties of k-d trees
  - a leaf has $\leq c$ points
  - an internal node has one point of its own plus one or two children
  - an internal node is split into two subspaces through its point
  - axis along which to split nodes are chosen cyclically (first along x-axis, then along y-axis, and so on)

Leaf:



Internal:





How to build a k-d tree

Possible strategies:

- an insertion-based method
  - define a method to add a single point into a tree
  - start from an empty tree and add all points into it

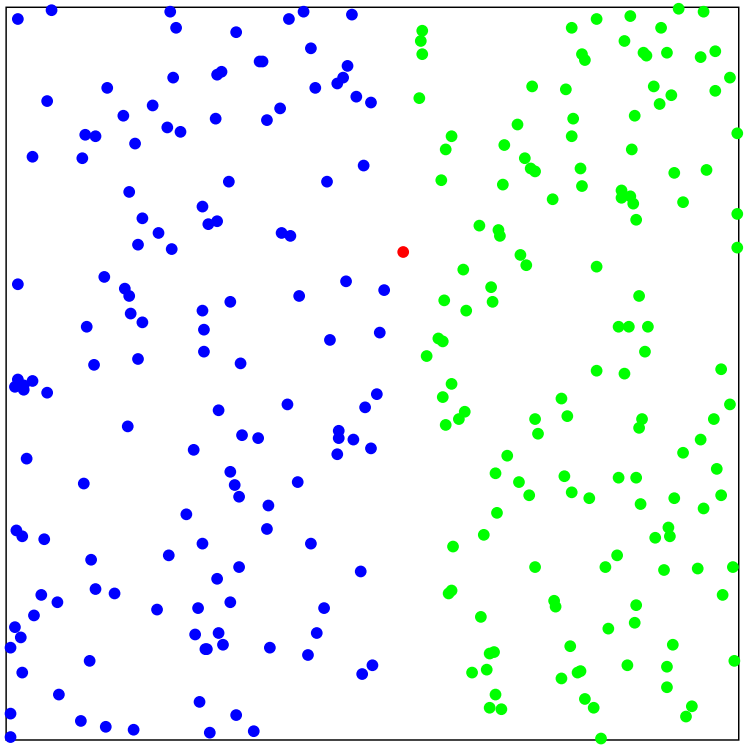
How to build a k-d tree

Possible strategies:

- an insertion-based method
  - define a method to add a single point into a tree
  - start from an empty tree and add all points into it
- a divide and conquer method

divide and conquer method

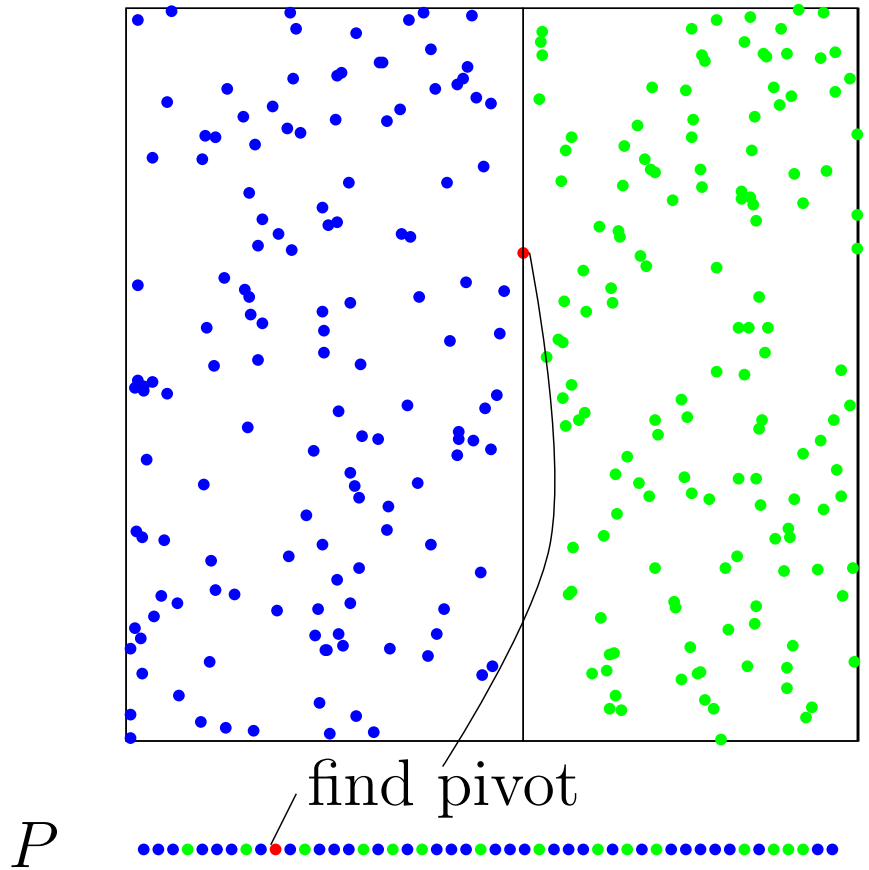
- to build a tree for a rectangle R and points P in R,



P

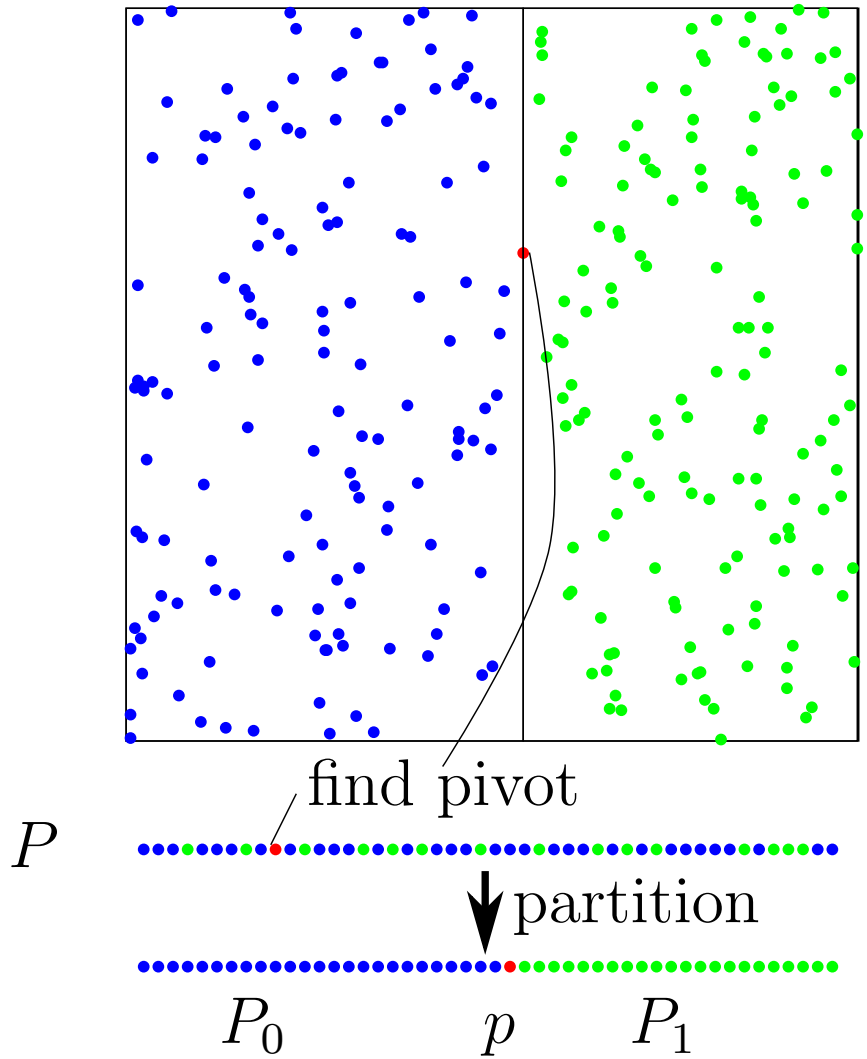
divide and conquer method

- to build a tree for a rectangle R and points P in R,
- choose a point $p \in P$ through which to split R, and



divide and conquer method

- to build a tree for a rectangle R and points P in R,
- choose a point $p \in P$ through which to split R, and
- partition P into $P_0 + \{p\} + P_1$
  - let's say we split along x-axis. then
  - P_0 : points whose x coodinate $< p$'s
  - P_1 : points whose x coodinate $\geq p$'s (except p)



divide and conquer method

```
1  /* build a k−d tree for a set of points P in a rectangular region R and return
2     the root of the tree. the node is at depth, so it should split along
3     (depth % D)th axis */
4  build(P, R, depth) {
5    if (|P| == 0) {
6      return 0; /* empty */
7    } else if (|P| <= threshold) {
8      /* small enough; leaf */
9      return make_leaf(P, R, depth);
10   } else {
11     /* find a point whose coordinate to split is near the median */
12     s = find_median(P, depth % D);
13     /* split R into two sub−rectangles */
14     R0,R1 = split_rect(R, depth % D, s.pos[depth % D]);
15     /* partition P by their coodinate lower/higher than p's coordinate */
16     P0,P1 = partition(P - { p }, depth % D, s.pos[depth % D]);
17     /* build a tree for each rectangle */
18     n0 = build(P0, R0, depth + 1);
19     n1 = build(P1, R1, depth + 1);
20     /* return a node having n0 and n1 as its children */
21     return make_node(p, n0, n1, depth);
22   }
23 }
```

Notes on subprocedures

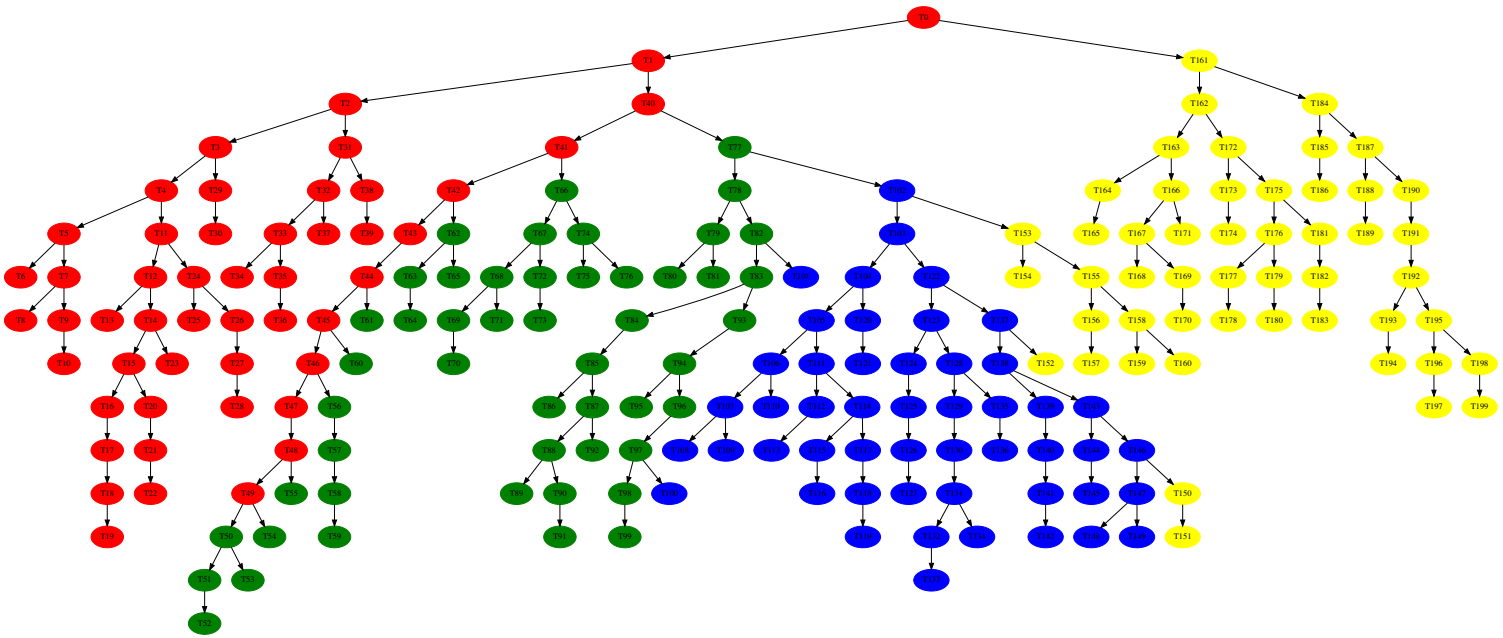
- $s =$ `find_median`(P, d)
  - find a point $\in P$ whose dth coordinate is (close to) the median value among all points in P
  - sample a few points and choose the median $\Rightarrow O(1)$
- $R_0, R_1 =$ `split_rect`(R, d, c)
  - split a rectangular region R by a (hyper-)plane "dth coordinate $= c$"
  - just make two rectangular regions $\Rightarrow O(1)$
- $P_0, P_1 =$ `partition`(P, d, c)
  - partition a set of points P into two subsets P_0 (dth coordinate $< c$) and P_1 (dth coordinate $\geq c$)
  - $\Rightarrow O(|P|)$

Contents

Contents

Parallelizing divide and conquer

- Divide and conquer algorithms are easy to parallelize if the programming language/library supports asynchronous recursive calls (*task parallel* systems)
  - OpenMP task constructs
  - Intel Threading Building Block (TBB)
  - Cilk, CilkPlus
- we use TBB in the following because of its performance portability



Parallelizing k-d tree construction with tasks

- it's as simple as doing two recursions in parallel!
- e.g., with OpenMP tasks

```
1  build(P, R, depth) {
2    if (|P| == 0) {
3      return 0; /* empty */
4    } else if (|P| <= threshold) {
5      return make_leaf(P, R, depth);
6    } else {
7      s = find_median(P, depth % D);
8      R0,R1 = split_rect(R, depth % D, s.pos[depth % D]);
9      P0,P1 = partition(P - { p }, depth % D, s.pos[depth % D]);
10 #pragma omp task shared(n0)
11     n0 = build(P0, R0, depth + 1);
12 #pragma omp task shared(n1)
13     n1 = build(P1, R1, depth + 1);
14 #pragma omp taskwait
15     return make_node(p, n0, n1, depth);
16   }
17 }
```

- do you want to parallelize it with only parallel loops?

Contents

Task parallelism in Intel TBB

- we use Intel TBB as a specific example of task parallel systems, primarily because of its compiler-independence
- consider executing the following two calls in parallel

```
1    n0 = build(P0, R0, depth + 1);
2    n1 = build(P1, R1, depth + 1);
```

- in TBB, this can be expressed by:

```
1    tbb::task_group tg;
2    tg.run([&] { n0 = build(P0, R0, depth + 1); });
3    n1 = build(P1, R1, depth + 1);
4    tg.wait();
```

- note: you can, but do not have to, create a task for the second recursion

Parallelizing k-d tree construction with TBB

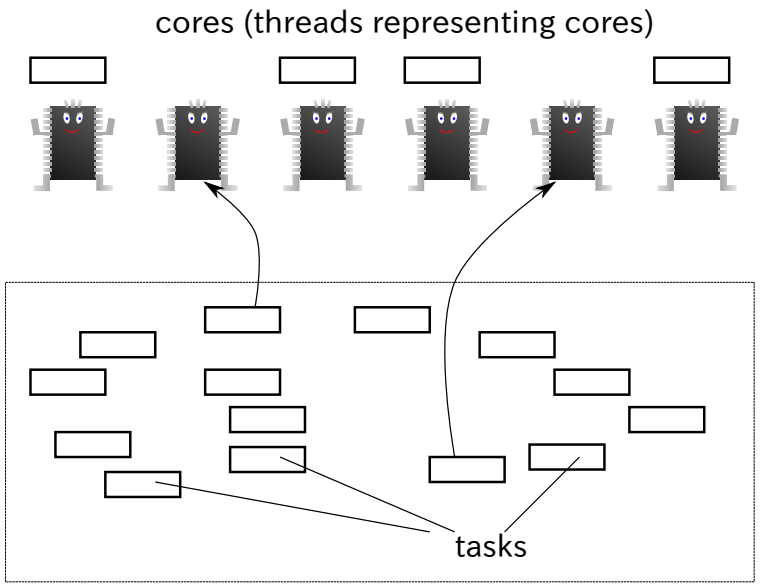
```
1  build(P, R, depth) {
2    if (|P| == 0) {
3      return 0; /* empty */
4    } else if (|P| <= threshold) {
5      return make_leaf(P, R, depth);
6    } else {
7      s = find_median(P, depth % D);
8      R0,R1 = split_rect(R, depth % D, s.pos[depth % D]);
9      P0,P1 = partition(P - { p }, depth % D, s.pos[depth % D]);
10     tbb::task_group tg;
11     tg.run([&] { n0 = build(P0, R0, depth + 1); });
12     n1 = build(P1, R1, depth + 1);
13     tg.wait();
14     return make_node(p, n0, n1, depth);
15   }
16 }
```

A summary of `task_group` class in TBB

- include `<tbb/task_group.h>`
- create an instance of `tbb::task_group` class prior to creating tasks
- `task_group::run(`c`)` method:
  - creates a task that executes $c()$
  - c is any *callable* object
- `task_group::wait()` method:
  - waits for the completion of all tasks that are `run`'ed by the `task_group` object

How TBB executes tasks?

- details are complex and will be covered later, but the following suffices for now as an approximation
  - it globally pools ready (runnable) tasks somewhere
  - from which cores fetch and execute tasks
- tasks can be executed by any core (load balancing)
- each core tries to fill it with tasks as much as possible (greedy)



A note on callable and C++ lambda expression

- any *callable* object can be passed to `run` method
- a callable object is an object defining `operator()`
- e.g.

```
1  class my_task {
2    int x;
3    my_task(int arg) { x = arg; }
4    void operator() () { print(x); }
5  }
```

```
1  tbb::task_group tg;
2  my_task c(3);
3  tg.run(c);
```

C++ lambda expression (closure)

- it is cumbersome to manually define a class for each task creation; C++ lambda expression reduces this labor
  - *not* specific to TBB, but a recent extension to C++ (C++0x, C++11) that TBB programmers would appreciate
- syntax in brief:

  [*capture-list*] { *statements* }

  this represents a callable object that, when called, executes *statements*.
- *capture-list* specifies which variables to *copy* into the closure, and which variables to *share* between the closure and the outer context

C++ lambda expression examples (1)

- copy everything (`x` and `y`);

```
1 int x = 3;
2 C = [=] { print(x); };
3 x = 4;
4 C(); // will print 3, not 4
```

- share everything

```
1 int x = 3;
2 C = [&] { print(x); };
3 x = 4;
4 C(); // will print 4, not 3
```

C++ lambda expression examples (2)

- share specific variables

```
1 int z;
2 C = [=,&z] { z = 4; };
3 C();
4 print(z); // will print 4, not 3
```

- you will (often) like to share large arrays to avoid large copies

```
1 int a[1000];
2 C = [=,&a] { gemm(a) };
3 C();
```

Which variables to share/copy when you `run`?

- if you want to get a return value, do not forget to share the receiving variable
  - presumably wrong

```
1 tg.run([=] { y = f(x); });
```

  - you should insead write:

```
1 tg.run([=,&y] { y = f(x); });
```

- make sure arguments you passed to the task are not overwritten by the parent
  - presumably wrong

```
1 for (i = 0; i < n; i++) {
2   tg.run([&] { y = f(i); });
3 }
```

  - you should insead write:

```
1 tg.run([&,=i] { y = f(i); });
```

Contents

Reasoning about speedup

- so you parallelized your program, you now hope to get some speedup on parallel machines!

Reasoning about speedup

- so you parallelized your program, you now hope to get some speedup on parallel machines!
- PROBLEM: how to reason about the execution time (thus speedup) of the program with P processors

Reasoning about speedup

- so you parallelized your program, you now hope to get some speedup on parallel machines!
- PROBLEM: how to reason about the execution time (thus speedup) of the program with P processors

- ANSWER: get the *work* and the *critical path length* of the computation

Contents

Work and critical path length

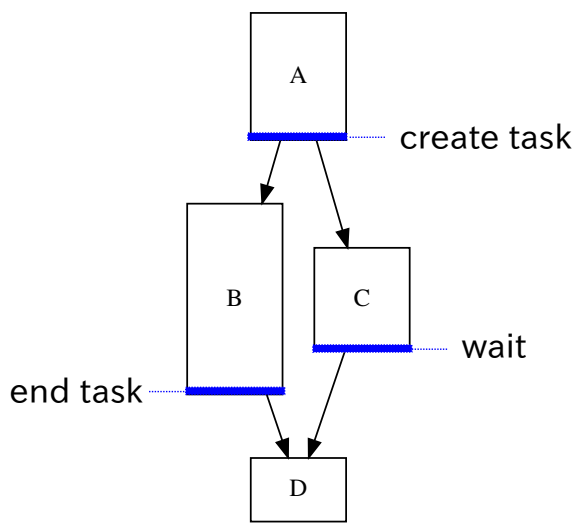
- **Work:** = the total amount of work of the computation
  - = the time it takes in a serial execution
- **Critical path length:** = the maximum length of dependent chain of computation
  - a more precise definition follows, with *computational DAGs*

Computational DAGs

The DAG of a computation is a directed acyclic graph in which:

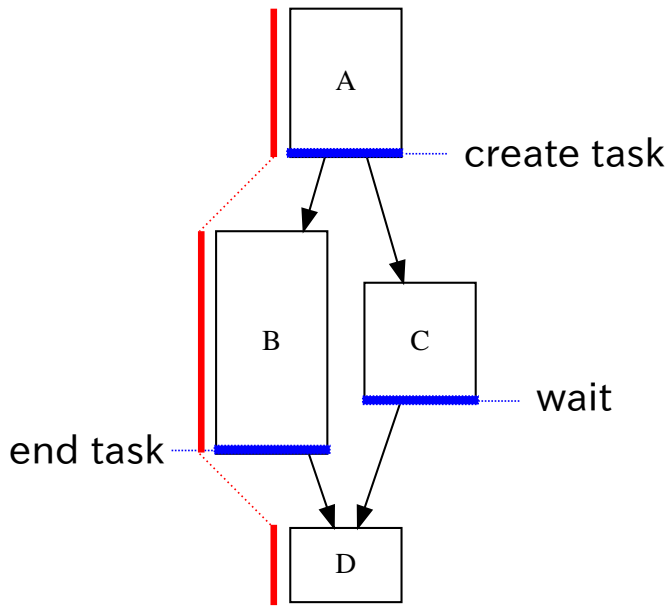
- a node = an interval of computation free of task parallel primitives
  - i.e. a node *starts* and *ends* by a task parallel primitive
  - we assume a single node is executed non-preemptively
- an edge = a dependency between two nodes, of three types:
  - parent → created child
  - child → waiting parent
  - a node → the next node in the same task

```
1 main() {
2   A();
3   create_task B();
4   C();
5   wait(); // wait for B
6   D();
7 }
```



A computational DAG and critical path length

- Consider each node is augmented with a time for a processor to execute it (*the node's execution time*)
- Define *the length of a path* to be the sum of execution time of the nodes on the path



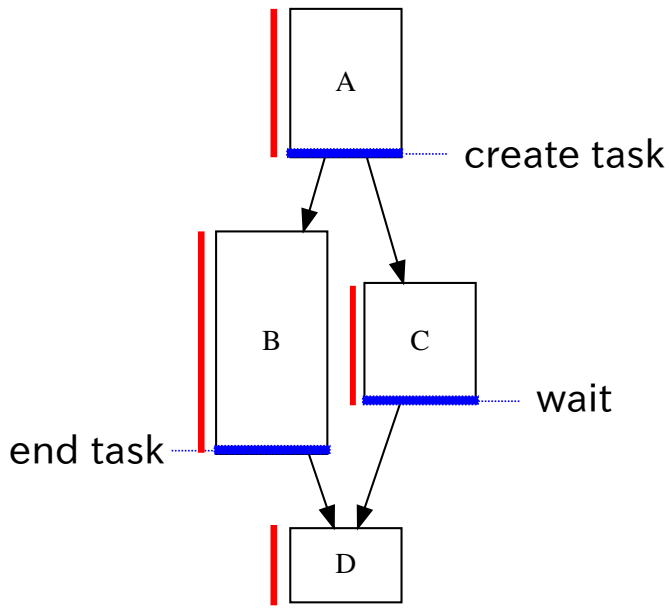
Given a computational DAG,

critical path length = the length of the longest paths from the start node to the end node in the DAG

(we often say *critical path* to in fact mean its length)

A computational DAG and work

- Work, too, can be elegantly defined in light of computational DAGs



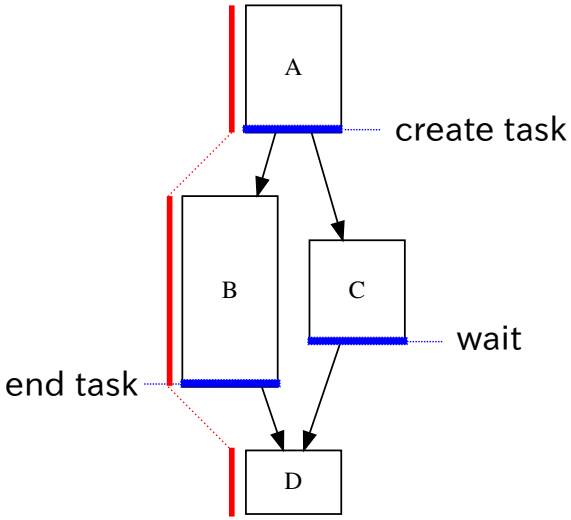
Given a computational DAG,

work = the sum of lengths of all nodes

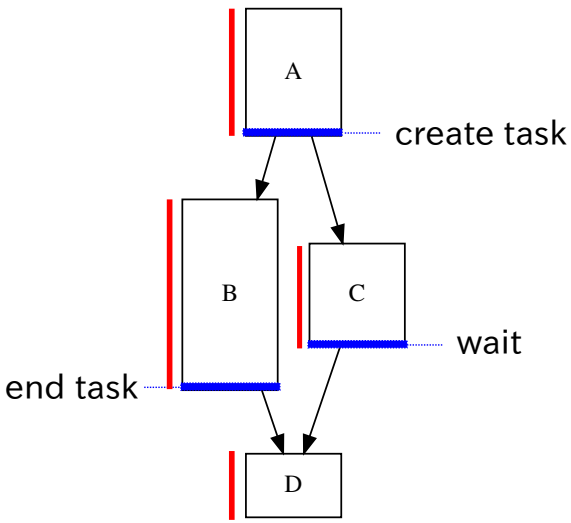
What do they intuitively mean?

- The critical path length represents the "ideal" execution time with *infinitely many* processors
  - i.e., each node is executed immediately after all its predecessors have finished
- We thus often denote it by T_∞
- Analogously, we often denote *work* by T_1

$T_1 = \textit{work}$, $T_\infty = \textit{critical path}$



critical path length



work

Why are they important?

- Now you understood what the critical path is

Why are they important?

- Now you understood what the critical path is
- But why is it a good tool to understand speedup?

Why are they important?

- Now you understood what the critical path is
- But why is it a good tool to understand speedup?

- QUESTION: Specifically, what does it tell us about performance or speedup on, say, my 64 core machines?

Why are they important?

- Now you understood what the critical path is
- But why is it a good tool to understand speedup?

- QUESTION: Specifically, what does it tell us about performance or speedup on, say, my 64 core machines?
- ANSWER: A beautiful theorem (*greedy scheduler theorem*) gives us an answer

Contents

The greedy scheduler theorem

- Assume:
  - you have P processors
  - they are *greedy*, in the sense that a processor is *always busy* on a task whenever there is *any* runnable task in the entire system
  - an execution time of a node does not depend on which processor executed it

The greedy scheduler theorem

- Assume:
  - you have P processors
  - they are *greedy*, in the sense that a processor is *always busy* on a task whenever there is *any* runnable task in the entire system
  - an execution time of a node does not depend on which processor executed it
- Theorem: given a computational DAG of:
  - work T_1 and
  - critical path T_∞,

  the execution time with P processors, T_P, satisfies

$$T_P \leq \frac{T_1 - T_\infty}{P} + T_\infty$$

The greedy scheduler theorem

- Assume:
  - you have P processors
  - they are *greedy*, in the sense that a processor is *always busy* on a task whenever there is *any* runnable task in the entire system
  - an execution time of a node does not depend on which processor executed it
- Theorem: given a computational DAG of:
  - work T_1 and
  - critical path T_∞,

  the execution time with P processors, T_P, satisfies

$$T_P \leq \frac{T_1 - T_\infty}{P} + T_\infty$$

- in practice you remember a simpler form:

$$T_P \leq \frac{T_1}{P} + T_\infty$$

The greedy scheduler theorem

- it is now a common sense in parallel computing, but the root of the idea seems:

  Richard Brent. *The Parallel Evaluation of General Arithmetic Expressions.* Journal of the ACM 21(2). pp201-206. 1974

  Derek Eager, John Zahorjan, and Edward Lazowska. *Speedup versus efficiency in parallel systems.* IEEE Transactions on Computers 38(3). pp408-423. 1989

- People attribute it to Brent and call it Brent's theorem
- Proof is a good exercise for you

I'll repeat! Remember it!

$$T_P \leq \frac{T_1}{P} + T_\infty$$

Facts around T_1 and T_∞

Consider the execution time with P processors (T_P)

- there are two obvious lower bounds
  - $T_P \geq \frac{T_1}{P}$
  - $T_P \geq T_\infty$
- what a greedy scheduler achieves is intriguing (the sum of two lower bounds)

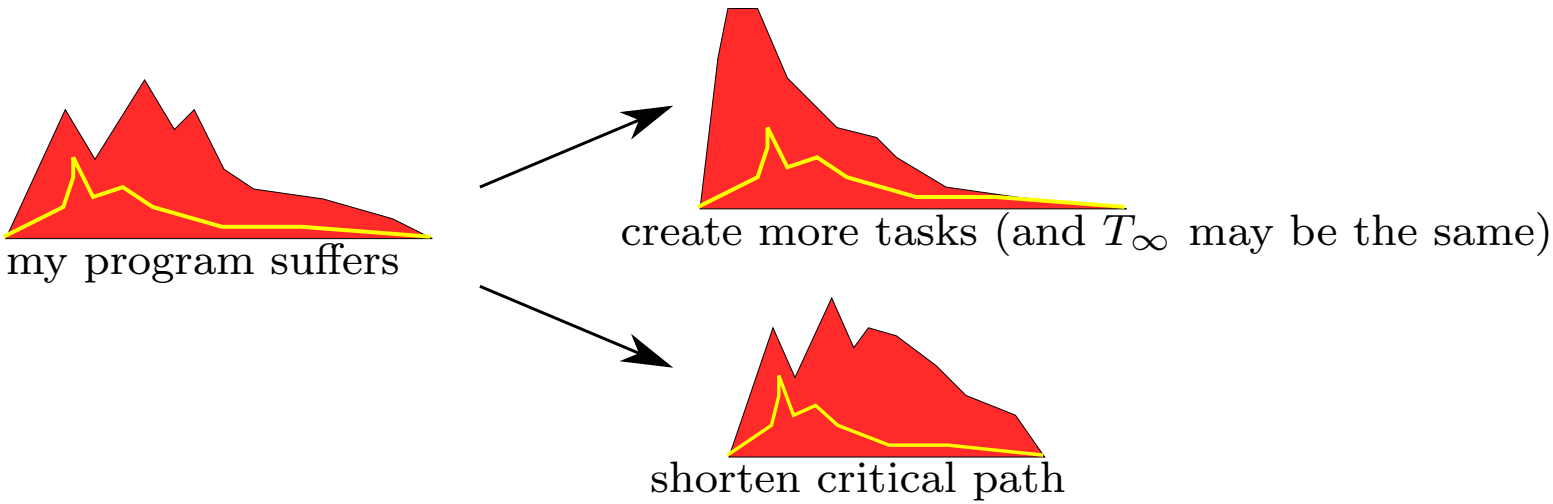
$$T_P \leq \frac{T_1}{P} + T_\infty$$

- if $T_1/T_\infty \gg P$, the second term is negligible ($\Rightarrow$ you get nearly perfect speedup)
- any greedy scheduler is within a factor of two of the optimal scheduler (下手な考え休むに似たり?)

Takeaway message

Suffer from low parallelism? ⇒ try to shorten its critical path

in contrast, people are tempted to get more speedup by creating more and more tasks; they are useless unless doing so shortens the critical path



What makes T_∞ so useful?

T_∞ is:

- a single *global metric* (just as the work is)
  - not something that fluctuates over time (cf. the number of tasks)
- *inherent to the algorithm, independent from the scheduler*
  - not something that depends on schedulers (cf. the number of tasks)
- connected to execution time with P processors in a beautiful way ($T_P \leq T_1/P + T_\infty$)
- easy to estimate/calculate (like the ordinary time complexity of serial programs)

Contents

Calculating work and critical path

- for recursive procedures, using recurrent equations is often a good strategy
- e.g., if we have

```
1  f(n) {
2    if (n == 1) { trivial(n); /* assume O(1) */ }
3    else {
4      g(n);
5      create_task f(n/3);
6      f(2*n/3);
7      wait();
8    }
9  }
```

  then
  - (work) $W_{\mathrm{f}}(n) \leq W_{\mathrm{g}}(n) + W_{\mathrm{f}}(n/3) + W_{\mathrm{f}}(2n/3)$
  - (critical path) $C_{\mathrm{f}}(n) \leq C_{\mathrm{g}}(n) + \max\{C_{\mathrm{f}}(n/3), C_{\mathrm{f}}(2n/3)\}$
- we apply this for programs we have seen

Work of k-d tree construction

```
1  build(P, R, depth) {
2    if (|P| == 0) {
3      return 0; /* empty */
4    } else if (|P| <= threshold) {
5      return make_leaf(P, R, depth);
6    } else {
7      s = find_median(P, depth % D);
8      R0,R1 = split_rect(R, depth % D, s.pos[depth % D]);
9      P0,P1 = partition(P - { p }, depth % D, s.pos[depth % D]);
10     n0 = create_task build(P0, R0, depth + 1);
11     n1 = build(P1, R1, depth + 1);
12     wait();
13     return make_node(p, n0, n1, depth);
14   }
15 }
```

$$W_{\text{build}}(n) \approx 2W_{\text{build}}(n/2) + \Theta(n)$$

omitting math,

$$\therefore W_{\text{build}}(n) \in \Theta(n \log n)$$

Remark

- the argument above is crude, as n points are not always split into two sets of equal sizes
- yet, the $\Theta(n \log n)$ result is valid, as long as a split is guaranteed to be "never too unbalanced" (i.e., there is a constant $\alpha <$, s.t. each child gets $\leq \alpha n$ points)

Critical path

```
1  build(P, R, depth) {
2    if (|P| == 0) {
3      return 0; /* empty */
4    } else if (|P| <= threshold) {
5      return make_leaf(P, R, depth);
6    } else {
7      s = find_median(P, depth % D);
8      R0,R1 = split_rect(R, depth % D, s.pos[depth % D]);
9      P0,P1 = partition(P - { p }, depth % D, s.pos[depth % D]);
10     n0 = create_task build(P0, R0, depth + 1);
11     n1 = build(P1, R1, depth + 1);
12     wait();
13     return make_node(p, n0, n1, depth);
14   }
15 }
```

$$C_{\text{build}}(n) \approx C_{\text{build}}(n/2) + \Theta(n)$$

omitting math,

$$\therefore C_{\text{build}}(n) \in \Theta(n)$$

Speedup of k-d tree construction

- Now we have:

$$W_{\text{build}}(n) \in \Theta(n \log n),$$
$$C_{\text{build}}(n) \in \Theta(n).$$

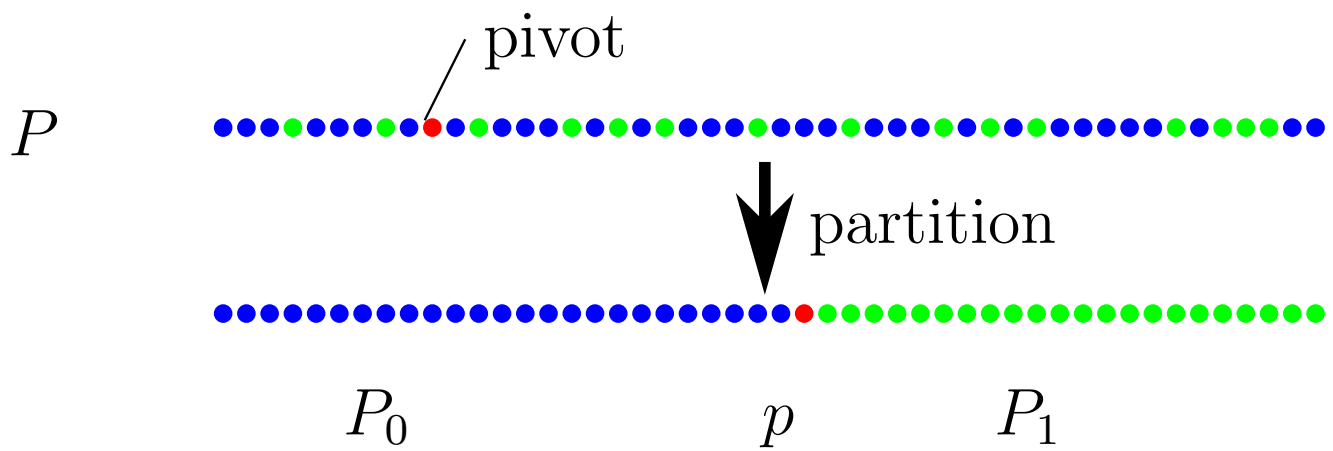
- $\Rightarrow$

$$\frac{T_1}{T_\infty} \in \Theta(\log n)$$

- not satisfactory in practice

What the analysis tells us

- the expected speedup, $\Theta(\log n)$, is not satisfactory
- to improve, shorten its critical path $\Theta(n)$, to $o(n)$
- where you should improve? the reason for the $\Theta(n)$ critical path is `partition`; we should parallelize `partition`
- this is left to the hands-on session (hint: divide and conquer thinking)



Contents

Two more divide and conquer examples

- Merge sort
- Cholesky factorization and its component matrix problems

Contents

Merge sort

- Input:
  - A: an array
- Output:
  - B: a sorted array
- Note: the result could be returned either in place or in a separate array. Assume it is "in place" in the following.

Merge sort : serial code

```
1  /* sort a..a_end and put the result into
2     (i)  a (if dest = 0)
3     (ii) t (if dest = 1) */
4  void ms(elem * a, elem * a_end,
5          elem * t, int dest) {
6    long n = a_end - a;
7    if (n == 1) {
8      if (dest) t[0] = a[0];
9    } else {
10     /* split the array into two */
11     long nh = n / 2;
12     elem * c = a + nh;
13     /* sort 1st half */
14     ms(a, c,     t,       1 - dest);
15     /* sort 2nd half */
16     ms(c, a_end, t + nh, 1 - dest);
17     elem * s = (dest ? a : t);
18     elem * d = (dest ? t : a);
19     /* merge them */
20     merge(s,       s + nh,
21           s + nh, s + n, d);
22   }
23 }
```

```
1  /* merge a_beg ... a_end
2       and b_beg ... b_end
3     into c */
4  void
5  merge(elem * a, elem * a_end,
6        elem * b, elem * b_end,
7        elem * c) {
8    elem * p = a, * q = b, * r = c;
9    while (p < a_end && q < b_end) {
10     if (*p < *q) { *r++ = *p++; }
11     else { *r++ = *q++; }
12   }
13   while (p < a_end) *r++ = *p++;
14   while (q < b_end) *r++ = *q++;
15 }
```

note: as always, actually switch to serial sort below a threshold (not shown in the code above)

Merge sort : parallelization

```
void ms(elem * a, elem * a_end,
        elem * t, int dest) {
  long n = a_end - a;
  if (n == 1) {
    if (dest) t[0] = a[0];
  } else {
    /* split the array into two */
    long nh = n / 2;
    elem * c = a + nh;
    /* sort 1st half */
    create_task ms(a, c,     t,      1 - dest);
    /* sort 2nd half */
    ms(c, a_end, t + nh, 1 - dest);
    wait();
    elem * s = (dest ? a : t);
    elem * d = (dest ? t : a);
    /* merge them */
    merge(s,      s + nh,
          s + nh, s + n, d);
  }
}
```

- Will we get "good enough" speedup?

Work of merge sort

```
void ms(elem * a, elem * a_end,
        elem * t, int dest) {
  long n = a_end - a;
  if (n == 1) {
    if (dest) t[0] = a[0];
  } else {
    /* split the array into two */
    long nh = n / 2;
    elem * c = a + nh;
    /* sort 1st half */
    create_task ms(a, c, t, 1 - dest);
    /* sort 2nd half */
    ms(c, a_end, t + nh, 1 - dest);
    wait();
    elem * s = (dest ? a : t);
    elem * d = (dest ? t : a);
    /* merge them */
    merge(s,      s + nh,
          s + nh, s + n, d);
  }
}
```

$$W_{\text{ms}}(n) = 2W_{\text{ms}}(n/2) + W_{\text{merge}}(n),$$
$$W_{\text{merge}}(n) \in \Theta(n).$$

$$\therefore W_{\text{ms}}(n) \in \Theta(n \log n)$$

Critical path of merge sort

```
void ms(elem * a, elem * a_end,
        elem * t, int dest) {
  long n = a_end - a;
  if (n == 1) {
    if (dest) t[0] = a[0];
  } else {
    /* split the array into two */
    long nh = n / 2;
    elem * c = a + nh;
    /* sort 1st half */
    create_task ms(a, c, t, 1 - dest);
    /* sort 2nd half */
    ms(c, a_end, t + nh, 1 - dest);
    wait();
    elem * s = (dest ? a : t);
    elem * d = (dest ? t : a);
    /* merge them */
    merge(s,      s + nh,
          s + nh, s + n, d);
  }
}
```

$$C_{\text{ms}}(n) = C_{\text{ms}}(n/2) + C_{\text{merge}}(n),$$
$$C_{\text{merge}}(n) \in \Theta(n)$$

$$\therefore C_{\text{ms}}(n) \in \Theta(n)$$

Speedup of merge sort

$$W_{\text{ms}}(n) \in \Theta(n \log n), C_{\text{ms}}(n) \in \Theta(n).$$

Since

$$T_P \geq C_{\text{ms}}(n),$$

the speedup (T_1/T_P) is

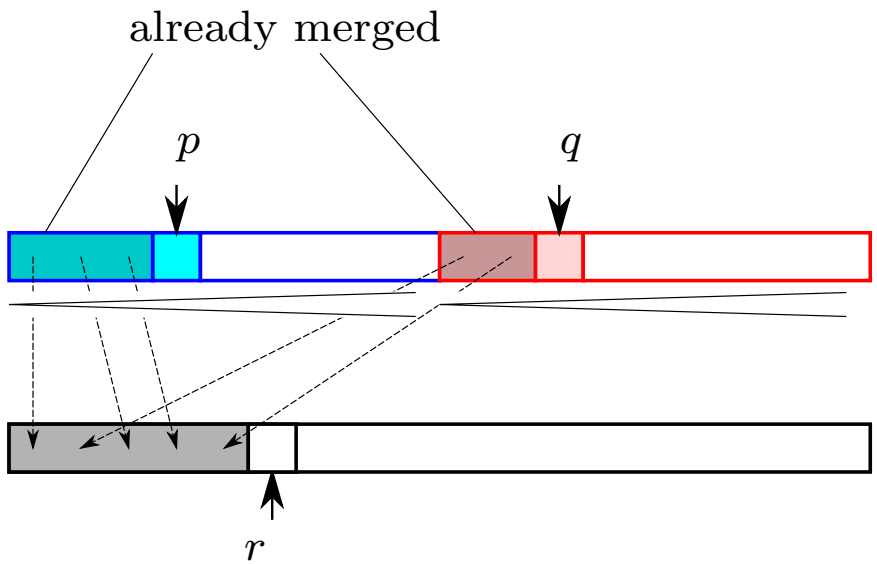
$$T_1/T_P \approx \Theta(\log n).$$

Visualizing parallelism

- Clearly, the "the last merge step" causes $\Theta(n)$ serial work
- We must have a parallel merge with $o(n)$ critical path

How (serial) merge works

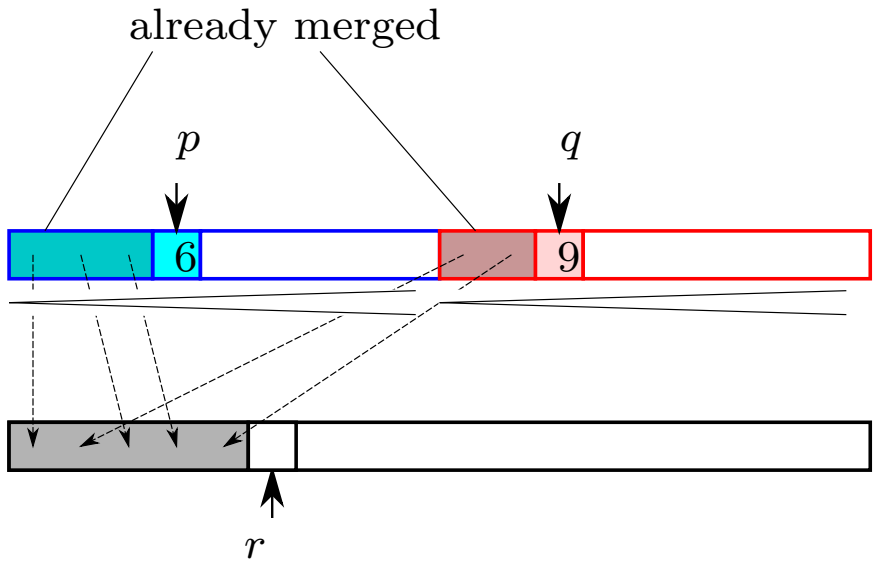
```
/* merge a_beg ... a_end
     and b_beg ... b_end
     into c */
void
merge(elem * a, elem * a_end,
      elem * b, elem * b_end,
      elem * c) {
  elem * p = a, * q = b, * r = c;
  while (p < a_end && q < b_end) {
    if (*p < *q) { *r++ = *p++; }
    else { *r++ = *q++; }
  }
  while (p < a_end) *r++ = *p++;
  while (q < b_end) *r++ = *q++;
}
```



Looks very serial . . .

How (serial) merge works

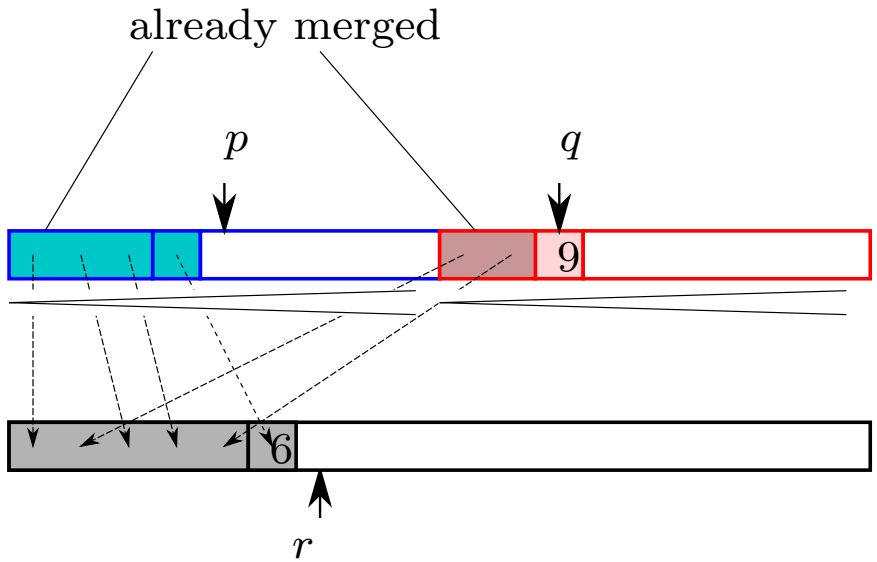
```
/* merge a_beg ... a_end
     and b_beg ... b_end
    into c */
void
merge(elem * a, elem * a_end,
      elem * b, elem * b_end,
      elem * c) {
  elem * p = a, * q = b, * r = c;
  while (p < a_end && q < b_end) {
    if (*p < *q) { *r++ = *p++; }
    else { *r++ = *q++; }
  }
  while (p < a_end) *r++ = *p++;
  while (q < b_end) *r++ = *q++;
}
```



Looks very serial . . .

How (serial) merge works

```
/* merge a_beg ... a_end
      and b_beg ... b_end
     into c */
void
merge(elem * a, elem * a_end,
      elem * b, elem * b_end,
      elem * c) {
  elem * p = a, * q = b, * r = c;
  while (p < a_end && q < b_end) {
    if (*p < *q) { *r++ = *p++; }
    else { *r++ = *q++; }
  }
  while (p < a_end) *r++ = *p++;
  while (q < b_end) *r++ = *q++;
}
```



Looks very serial . . .

How to parallelize merge?

- Again, divide and conquer thinking helps
- Hold the solution until the next hands-on

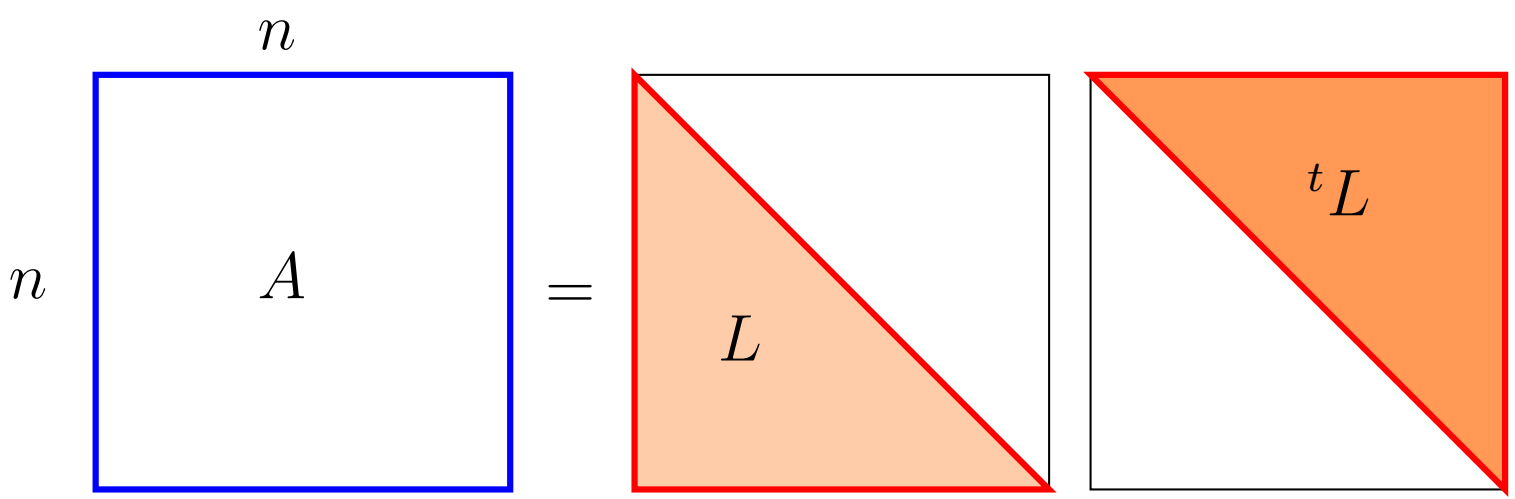
Contents

Our running example : Cholesky factorization

- Input:
  - A: $n \times n$ positive semidefinite symmetric matrix
- Output:
  - L: $n \times n$ lower triangular matrix s.t.

$$A = L\,{}^tL$$

  - (${}^tL$ is a transpose of L)



Note : why Cholesky factorization is important?

- It is the core step when solving

$$Ax = b \quad \text{(single righthand side)}$$

or, in more general,

$$AX = B \quad \text{(multiple righthand sides)},$$

as follows.

1. Cholesky decompose $A = L\,{}^tL$ and get

$$L \underbrace{{}^tLX}_{Y} = B$$

2. Find X by solving triangular systems twice
   1. $LY = B$
   2. ${}^tLX = Y$

Formulate using subproblems

$$\begin{pmatrix} A_{11} & {}^tA_{21} \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} L_{11} & O \\ L_{21} & L_{22} \end{pmatrix} \begin{pmatrix} {}^tL_{11} & {}^tL_{21} \\ O & {}^tL_{22} \end{pmatrix}$$

leads to three subproblems

1. $A_{11} = L_{11}\,{}^tL_{11}$
2. ${}^tA_{21} = L_{11}\,{}^tL_{21}$
3. $A_{22} = L_{21}\,{}^tL_{21} + L_{22}\,{}^tL_{22}$

Solving with recursions

$$\begin{pmatrix} A_{11} & {}^tA_{21} \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} L_{11} & O \\ L_{21} & L_{22} \end{pmatrix} \begin{pmatrix} {}^tL_{11} & {}^tL_{21} \\ O & {}^tL_{22} \end{pmatrix}$$

1. $A_{11} = L_{11}\,{}^tL_{11}$
2. ${}^tA_{21} = L_{11}\,{}^tL_{21}$
3. $A_{22} = L_{21}{}^tL_{21} + L_{22}\,{}^tL_{22}$

```
1  /* Cholesky factorization */
2  chol(A) {
3    if (n = 1) return (√a11);
4    else {
5      L11 = chol(A11);
6      /* triangular solve */
7      tL21 = trsm(L11, tA21);
8      L22 = chol(A22 − L21 tL21);
9      return ( L11   tL21 )
              ( L21   L22 )
10   }
11 }
```

Solving with recursions

$$\begin{pmatrix} A_{11} & {}^tA_{21} \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} L_{11} & O \\ L_{21} & L_{22} \end{pmatrix} \begin{pmatrix} {}^tL_{11} & {}^tL_{21} \\ O & {}^tL_{22} \end{pmatrix}$$

1. $A_{11} = L_{11}\,{}^tL_{11}$
   - recursion and get L_{11}
2. ${}^tA_{21} = L_{11}\,{}^tL_{21}$
3. $A_{22} = L_{21}{}^tL_{21} + L_{22}\,{}^tL_{22}$

```
1  /* Cholesky factorization */
2  chol(A) {
3    if (n = 1) return (√a11);
4    else {
5      L11 = chol(A11);
6      /* triangular solve */
7      tL21 = trsm(L11, tA21);
8      L22 = chol(A22 − L21 tL21);
9      return ( L11   tL21 )
              ( L21   L22 )
10   }
11 }
```

Solving with recursions

$$\begin{pmatrix} A_{11} & {}^tA_{21} \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} L_{11} & O \\ L_{21} & L_{22} \end{pmatrix} \begin{pmatrix} {}^tL_{11} & {}^tL_{21} \\ O & {}^tL_{22} \end{pmatrix}$$

1. $A_{11} = L_{11}\,{}^tL_{11}$
   - recursion and get L_{11}
2. ${}^tA_{21} = L_{11}\,{}^tL_{21}$
   - solve a *triangular* system and get ${}^tL_{21}$
3. $A_{22} = L_{21}{}^tL_{21} + L_{22}\,{}^tL_{22}$

```
1  /* Cholesky factorization */
2  chol(A) {
3    if (n = 1) return (√a11);
4    else {
5      L11 = chol(A11);
6      /* triangular solve */
7      tL21 = trsm(L11, tA21);
8      L22 = chol(A22 − L21 tL21);
9      return ( L11   tL21 )
              ( L21   L22 )
10   }
11 }
```

Solving with recursions

$$\begin{pmatrix} A_{11} & {}^tA_{21} \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} L_{11} & O \\ L_{21} & L_{22} \end{pmatrix} \begin{pmatrix} {}^tL_{11} & {}^tL_{21} \\ O & {}^tL_{22} \end{pmatrix}$$

1. $A_{11} = L_{11}\,{}^tL_{11}$
   - recursion and get L_{11}
2. ${}^tA_{21} = L_{11}\,{}^tL_{21}$
   - solve a *triangular* system and get ${}^tL_{21}$
3. $A_{22} = L_{21}\,{}^tL_{21} + L_{22}\,{}^tL_{22}$
   - recursion on $(A_{22} - L_{21}\,{}^tL_{21})$ and get L_{22}

```
1  /* Cholesky factorization */
2  chol(A) {
3    if (n = 1) return (√a11);
4    else {
5      L11 = chol(A11);
6      /* triangular solve */
7      tL21 = trsm(L11, tA21);
8      L22 = chol(A22 − L21 tL21);
9      return ( L11   tL21 )
              ( L21   L22 )
10   }
11 }
```

Remark 1 : "In-place update" version

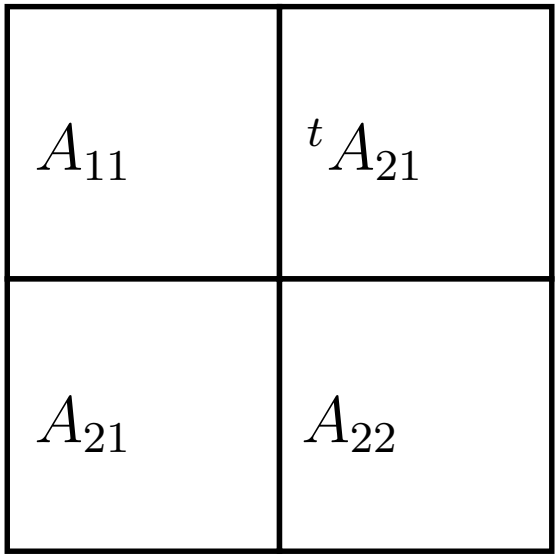
- Instead of returning the answer as another matrix, it is often possible to update the input matrix with the answer
- When possible, it is desirable, as it avoids extra copies

```
/* functional */                                          1
chol(A) {                                                 2
  if (n = 1) return (√a11);                               3
  else {                                                  4
    L11 = chol(A11);                                      5
    /* triangular solve */                                6
    tL21 = trsm(L11, tA21);                               7
    L22 = chol(A22 − L21 tL21);                           8
    return ( L11   tL21 )                                 9
           ( L21   L22  )                                 10
  }                                                       11
}                                                         12
```

```
/* in place */
chol(A) {
  if (n = 1) a11 := √a11;
  else {
    chol(A11);
    /* triangular solve */
    trsm(A11, A12);
    A21 = tA12;
    A22 -= A21 A12
    chol(A22);
  }
}
```

In-place Cholesky at work

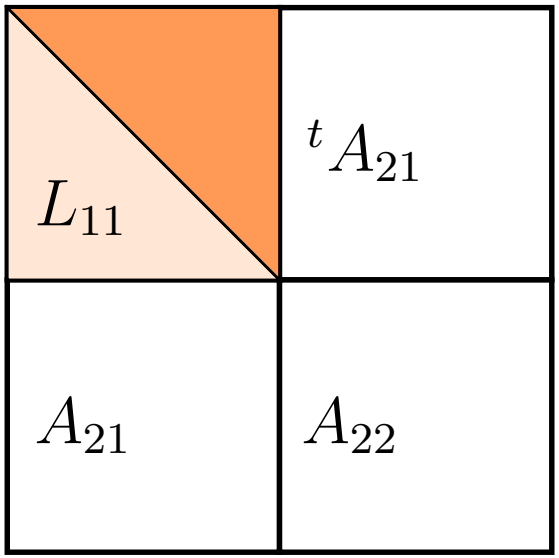
```
/* in place */
chol(A) {
  if (n = 1) a11 := √a11;
  else {
    chol(A11);
    /* triangular solve */
    trsm(A11, A12);
    A21 = tA12;
    A22 -= A21 A12
    chol(A22);
  }
}
```



In-place Cholesky at work

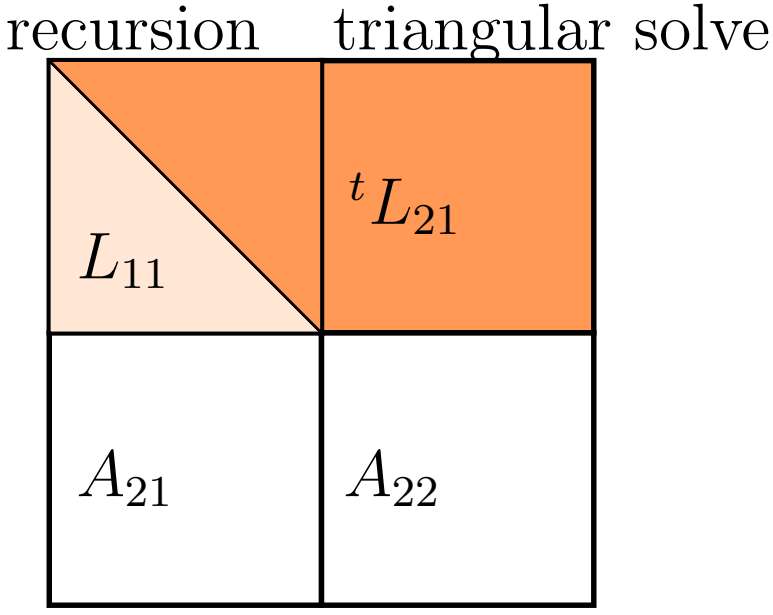
```
/* in place */
chol(A) {
  if (n = 1) a11 := √a11;
  else {
    chol(A11);
    /* triangular solve */
    trsm(A11, A12);
    A21 = tA12;
    A22 -= A21 A12
    chol(A22);
  }
}
```

recursion



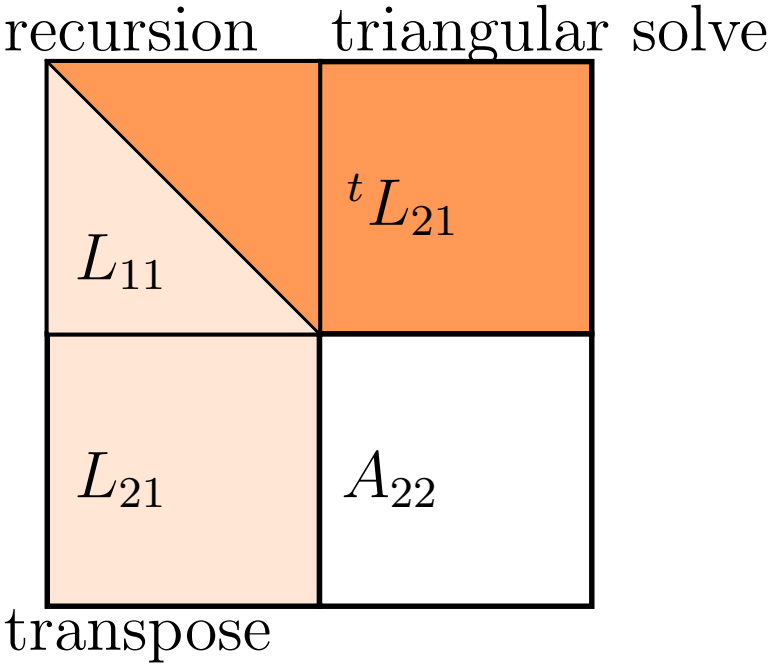
In-place Cholesky at work

```
/* in place */
chol(A) {
  if (n = 1) a11 := √a11;
  else {
    chol(A11);
    /* triangular solve */
    trsm(A11, A12);
    A21 = tA12;
    A22 -= A21 A12
    chol(A22);
  }
}
```



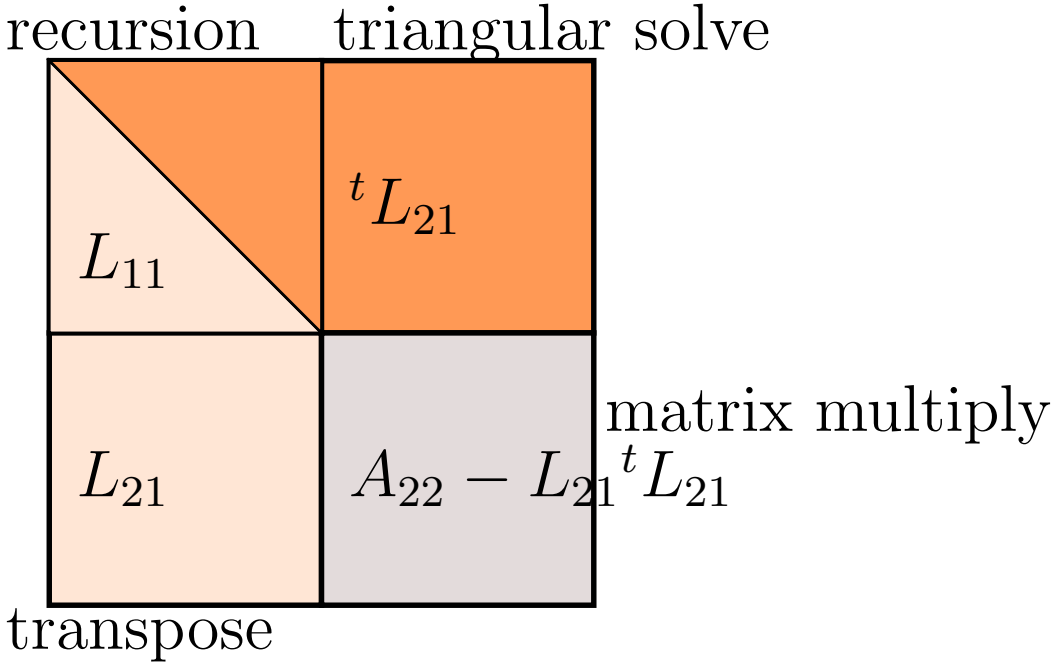
In-place Cholesky at work

```
/* in place */
chol(A) {
  if (n = 1) a11 := √a11;
  else {
    chol(A11);
    /* triangular solve */
    trsm(A11, A12);
    A21 = tA12;
    A22 -= A21 A12
    chol(A22);
  }
}
```



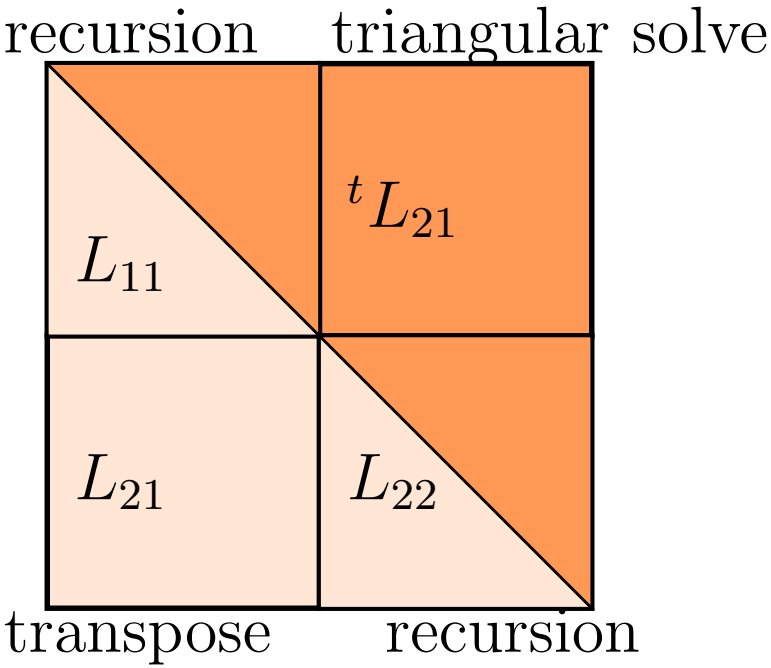
In-place Cholesky at work

```
/* in place */
chol(A) {
  if (n = 1) a11 := √a11;
  else {
    chol(A11);
    /* triangular solve */
    trsm(A11, A12);
    A21 = tA12;
    A22 -= A21A12
    chol(A22);
  }
}
```



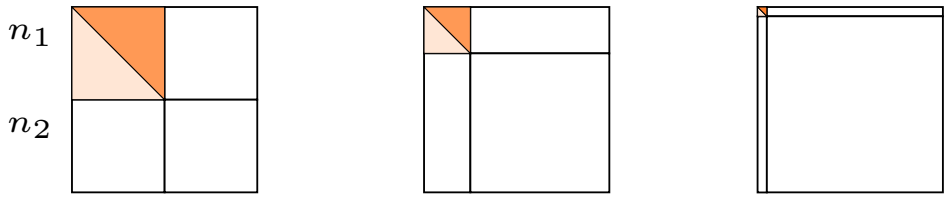
In-place Cholesky at work

```
/* in place */
chol(A) {
  if (n = 1) a11 := √a11;
  else {
    chol(A11);
    /* triangular solve */
    trsm(A11, A12);
    A21 = tA12;
    A22 -= A21A12
    chol(A22);
  }
}
```



Remark 2 : where to decompose

- Where to partition A is *arbitrary*
- The case $n_1 = 1$ and $n_2 = n - 1 \approx$ loops

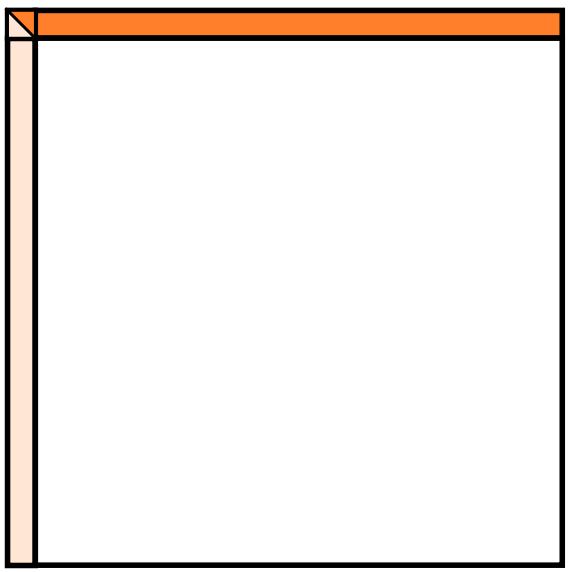



```
/* general */                          1
chol(A) {                              2
  if (n = 1) a11 := √a11;              3
  else {                               4
    chol(A11);                         5
    /* triangular solve */             6
    trsm(A11, A12);                    7
    A21 = tA12;                        8
    A22 -= A21A12                      9
    chol(A22);                         10
  }                                    11
}                                      12
```

```
/* loop-like */
chol(A) {
  if (n = 1) a11 := √a11;
  else {
    a11 := √a11;
    /* triangular solve */
    ā12 /= a11;
    ā21 /= a11;
    A22 -= ā21ā12
    chol(A22);
  }
}
```

Recursion to loops

- The "loop-like" version (partition into $1 + (n-1)$) can be written in a true loop

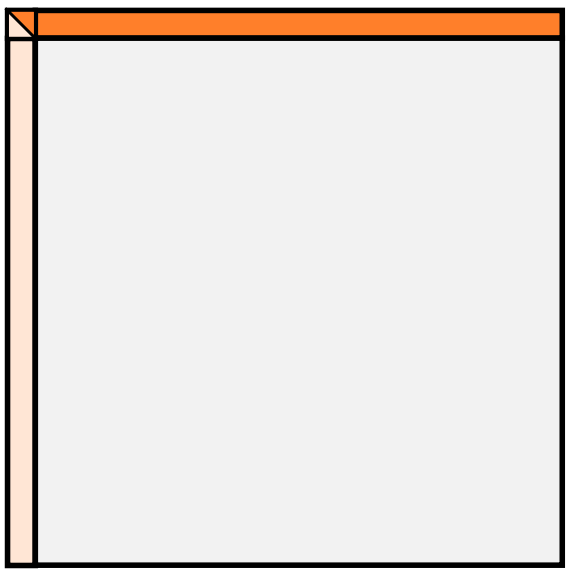


```
1  /* loop version */
2  chol_loop(A) {
3    for (k = 1; k ≤ n; k ++ ) {
4      a_kk := √a_kk;
5      A_{k,k+1:n} /= a_kk;
6      A_{k+1:n,k} /= a_kk;
7      A_{k+1:n,k+1:n} -= A_{k:n,k} A_{k,k:n}
8    }
9  }
```

In practice, you still need to code the loop to avoid creating too small tasks

Recursion to loops

- The "loop-like" version (partition into $1 + (n-1)$) can be written in a true loop

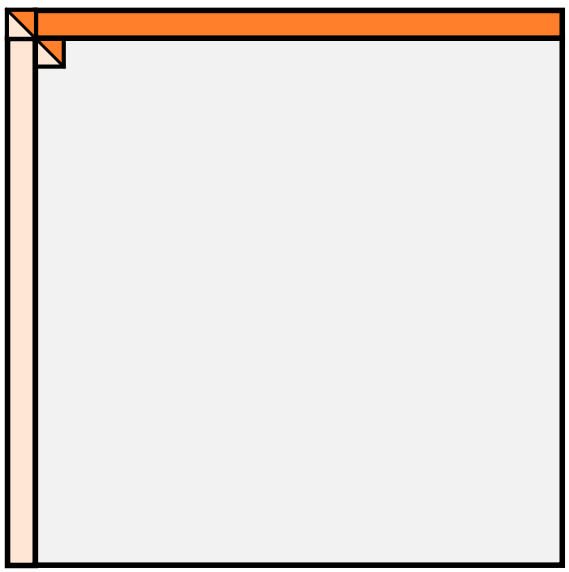


```
1  /* loop version */
2  chol_loop(A) {
3    for (k = 1; k ≤ n; k ++ ) {
4      a_kk := √a_kk;
5      A_{k,k+1:n} /= a_kk;
6      A_{k+1:n,k} /= a_kk;
7      A_{k+1:n,k+1:n} -= A_{k:n,k} A_{k,k:n}
8    }
9  }
```

In practice, you still need to code the loop to avoid creating too small tasks

Recursion to loops

- The "loop-like" version (partition into $1 + (n-1)$) can be written in a true loop

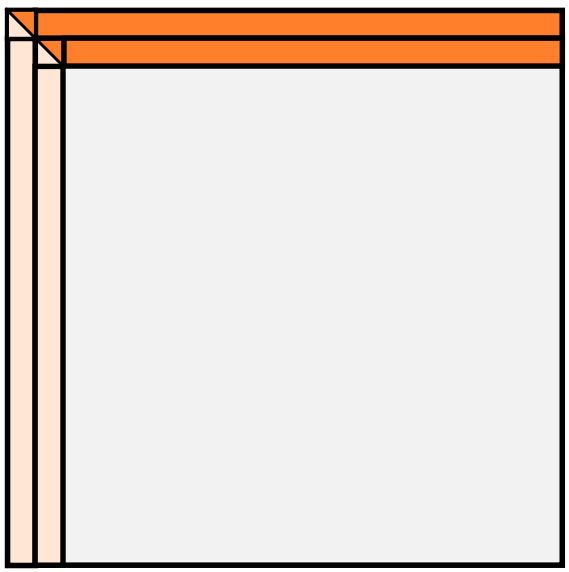


```
1  /* loop version */
2  chol_loop(A) {
3    for (k = 1; k ≤ n; k ++ ) {
4      a_kk := √a_kk;
5      A_{k,k+1:n} /= a_kk;
6      A_{k+1:n,k} /= a_kk;
7      A_{k+1:n,k+1:n} -= A_{k:n,k} A_{k,k:n}
8    }
9  }
```

In practice, you still need to code the loop to avoid creating too small tasks

Recursion to loops

- The "loop-like" version (partition into $1 + (n-1)$) can be written in a true loop

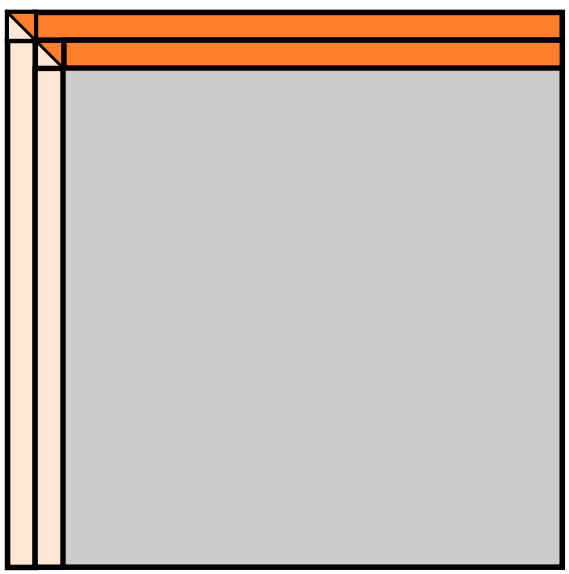


```
1  /* loop version */
2  chol_loop(A) {
3    for (k = 1; k ≤ n; k ++ ) {
4      a_kk := √a_kk;
5      A_{k,k+1:n} /= a_kk;
6      A_{k+1:n,k} /= a_kk;
7      A_{k+1:n,k+1:n} -= A_{k:n,k} A_{k,k:n}
8    }
9  }
```

In practice, you still need to code the loop to avoid creating too small tasks

Recursion to loops

- The "loop-like" version (partition into $1 + (n-1)$) can be written in a true loop

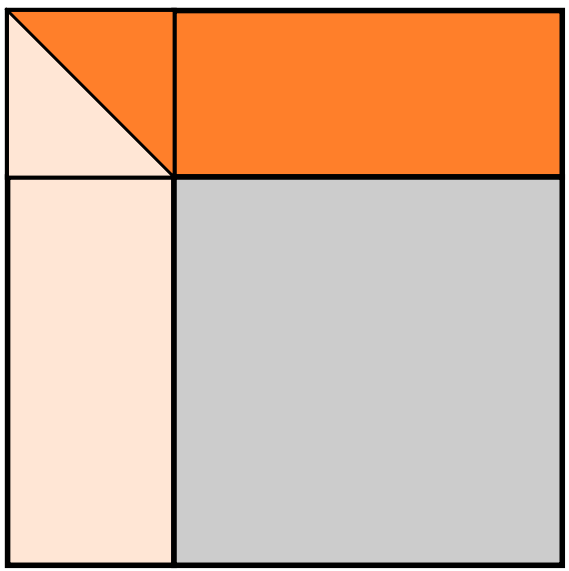


```
1  /* loop version */
2  chol_loop(A) {
3    for (k = 1; k ≤ n; k ++ ) {
4      a_kk := √a_kk;
5      A_{k,k+1:n} /= a_kk;
6      A_{k+1:n,k} /= a_kk;
7      A_{k+1:n,k+1:n} -= A_{k:n,k} A_{k,k:n}
8    }
9  }
```

In practice, you still need to code the loop to avoid creating too small tasks

Recursion to loops

- The "loop-like" version (partition into $1 + (n-1)$) can be written in a true loop

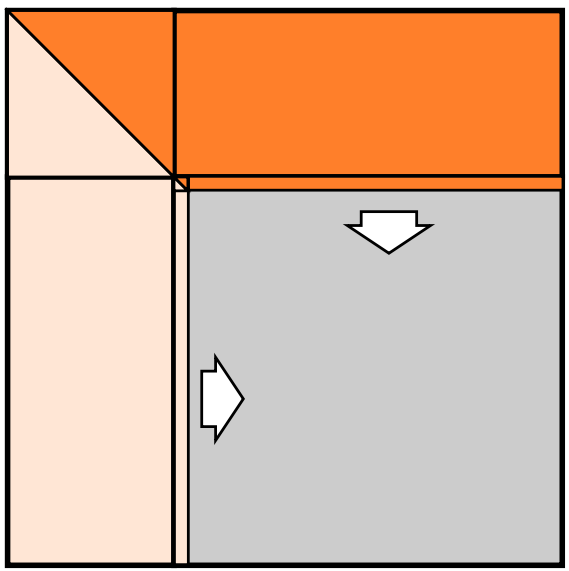


```
1  /* loop version */
2  chol_loop(A) {
3    for (k = 1; k ≤ n; k ++ ) {
4      a_kk := √a_kk;
5      A_{k,k+1:n} /= a_kk;
6      A_{k+1:n,k} /= a_kk;
7      A_{k+1:n,k+1:n} -= A_{k:n,k} A_{k,k:n}
8    }
9  }
```

In practice, you still need to code the loop to avoid creating too small tasks

Recursion to loops

- The "loop-like" version (partition into $1 + (n-1)$) can be written in a true loop



```
1  /* loop version */
2  chol_loop(A) {
3    for (k = 1; k ≤ n; k ++ ) {
4      a_kk := √a_kk;
5      A_{k,k+1:n} /= a_kk;
6      A_{k+1:n,k} /= a_kk;
7      A_{k+1:n,k+1:n} -= A_{k:n,k} A_{k,k:n}
8    }
9  }
```

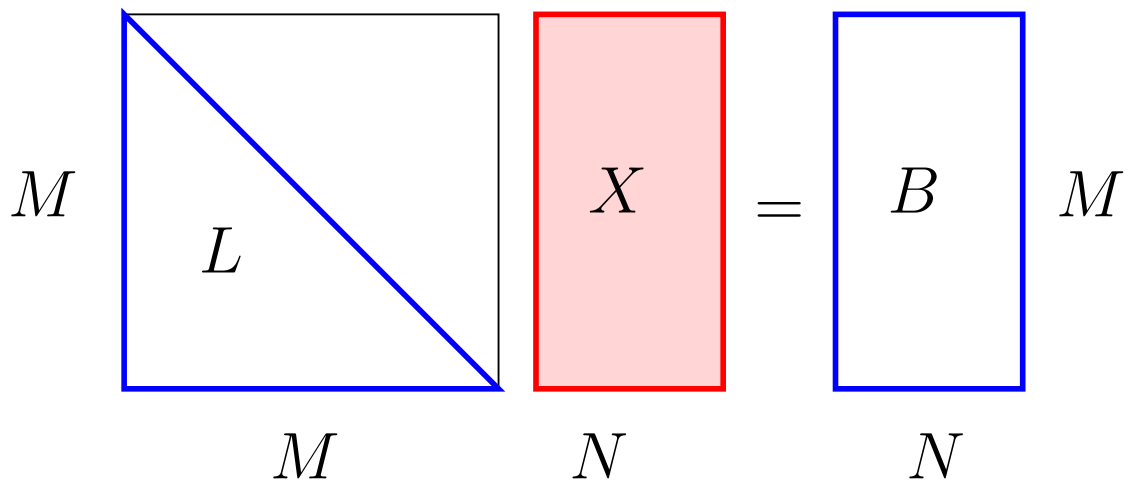
In practice, you still need to code the loop to avoid creating too small tasks

Contents

A subproblem 1: triangular solve

- Input:
  - L: $M \times M$ lower triangle matrix
  - B: $M \times N$ matrix
- Output:
  - X: $M \times N$ matrix X s.t.

$$LX = B$$



Formulate using subproblems

Two ways to decompose:

1. (split X and B vertically)

$$\begin{pmatrix} L_{11} & O \\ L_{21} & L_{22} \end{pmatrix} \begin{pmatrix} X_1 \\ X_2 \end{pmatrix} = \begin{pmatrix} B_1 \\ B_2 \end{pmatrix} \Rightarrow$$

   - $L_{11}X_1 = B_1$, and
   - $L_{21}X_1 + L_{22}X_2 = B_2$

2. (split X and B horizontally)

$$L \begin{pmatrix} X_1 & X_2 \end{pmatrix} = \begin{pmatrix} B_1 & B_2 \end{pmatrix} \Rightarrow$$

   - $LX_1 = B_1$, and
   - $LX_2 = B_2$

Choice is arbitrary, but for reasons we describe later, we decompose X and B so that their shapes are more square

Solving with recursions

1. $$\begin{pmatrix} L_{11} & O \\ L_{21} & L_{22} \end{pmatrix} \begin{pmatrix} X_1 \\ X_2 \end{pmatrix} = \begin{pmatrix} B_1 \\ B_2 \end{pmatrix}$$

   - $L_{11}X_1 = B_1$ recursion on (L_{11}, B_1) and get X_1
   - $L_{21}X_1 + L_{22}X_2 = B_2$ recursion on $(L_{22}, B_2 - L_{21}X_1)$ and get X_2

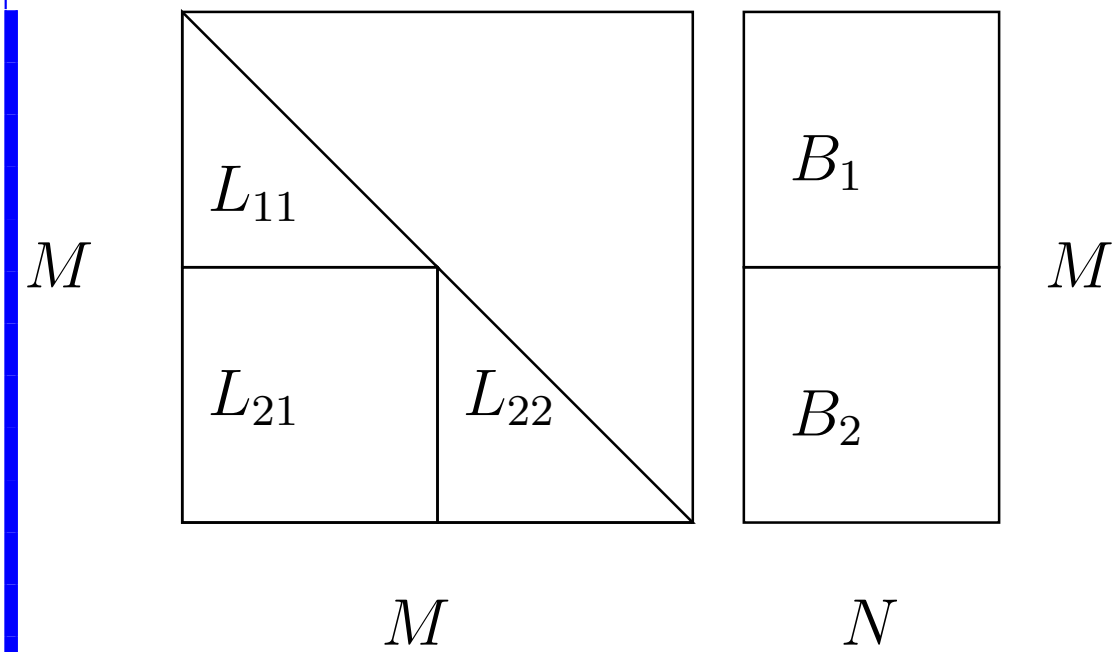
2. $$L \begin{pmatrix} X_1 & X_2 \end{pmatrix} = \begin{pmatrix} B_1 & B_2 \end{pmatrix} \Rightarrow$$

   solve them independently (easy)

```
1  /* triangular solve LX = B.
2     replace B with X */
3  trsm(L, B) {
4    if (M = 1) {
5      B /= l11;
6    } else if (M ≥ N) {
7      trsm(L11, B1);
8      B2 -= L21 B1;
9      trsm(L22, B2);
10   } else {
11     trsm(L, B1);
12     trsm(L, B2);
13   }
14 }
```

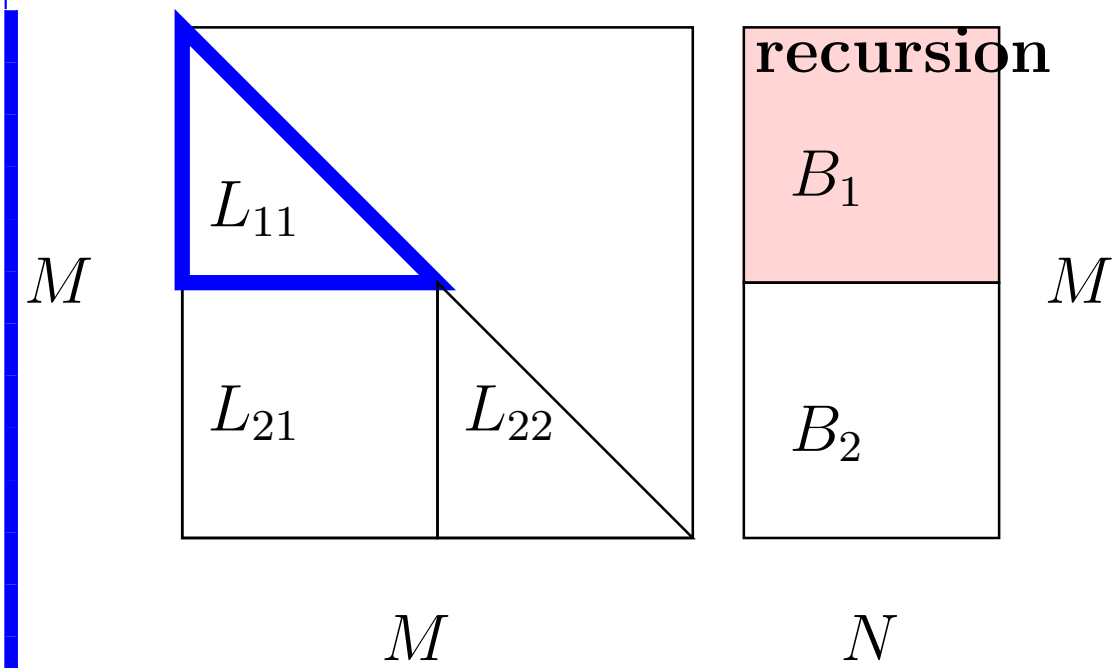
Triangular solve at work

```
/* triangular solve LX = B.
   replace B with X */
trsm(L, B) {
  if (M = 1) {
    B /= l11;
  } else if (M ≥ N) {
    trsm(L11, B1);
    B2 -= L21 B1;
    trsm(L22, B2);
  } else {
    trsm(L, B1);
    trsm(L, B2);
  }
}
```



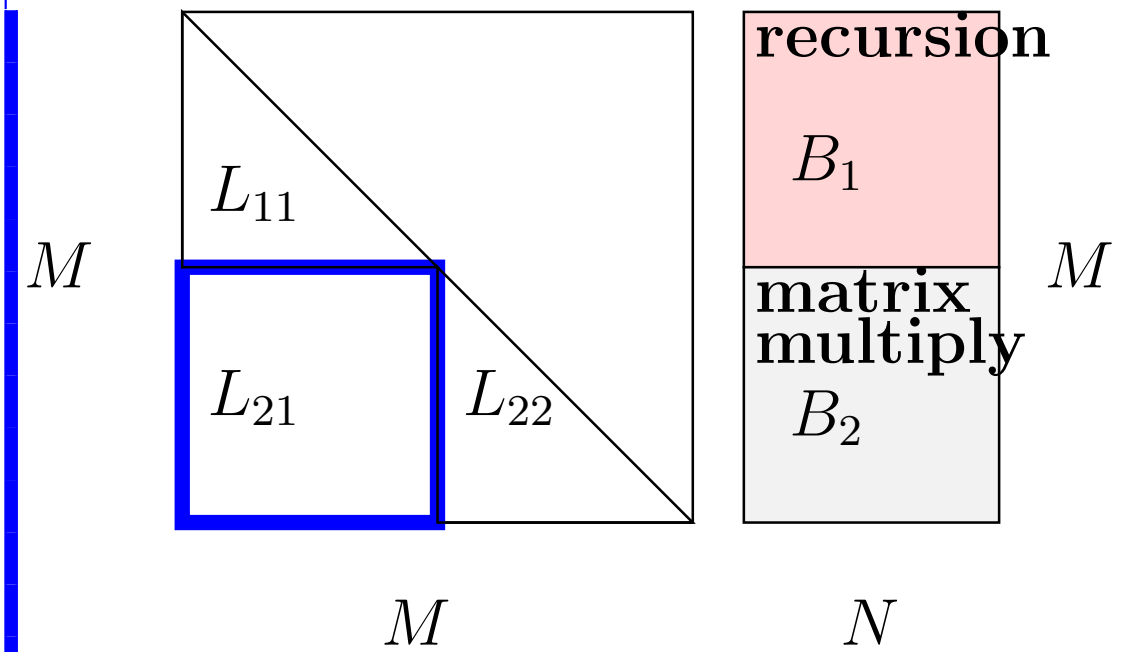
Triangular solve at work

```
/* triangular solve LX = B.
   replace B with X */
trsm(L, B) {
  if (M = 1) {
    B /= l11;
  } else if (M ≥ N) {
    trsm(L11, B1);
    B2 -= L21 B1;
    trsm(L22, B2);
  } else {
    trsm(L, B1);
    trsm(L, B2);
  }
}
```



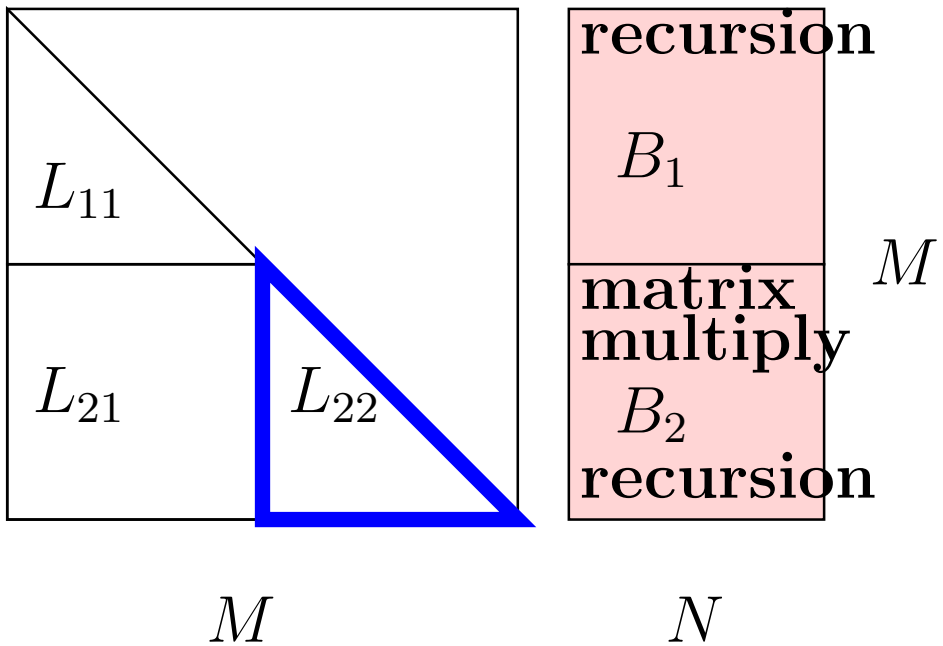
Triangular solve at work

```
/* triangular solve LX = B.
   replace B with X */
trsm(L, B) {
  if (M = 1) {
    B /= l11;
  } else if (M ≥ N) {
    trsm(L11, B1);
    B2 -= L21 B1;
    trsm(L22, B2);
  } else {
    trsm(L, B1);
    trsm(L, B2);
  }
}
```



Triangular solve at work

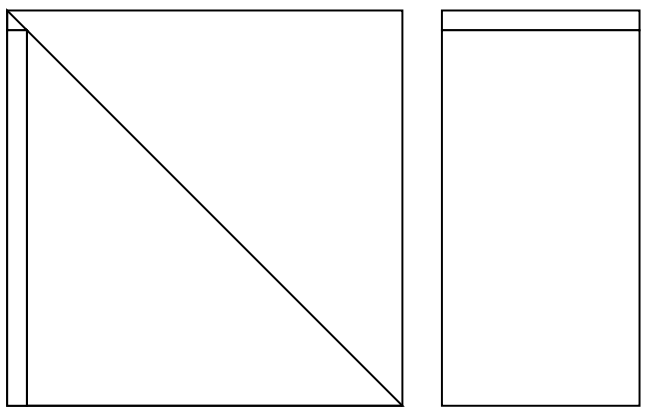
```
/* triangular solve LX = B.
   replace B with X */
trsm(L, B) {
  if (M = 1) {
    B /= l11;
  } else if (M ≥ N) {
    trsm(L11, B1);
    B2 -= L21 B1;
    trsm(L22, B2);
  } else {
    trsm(L, B1);
    trsm(L, B2);
  }
}
```



Recursions and loops

Again, partitioning is arbitrary and there is a loop-like partitioning

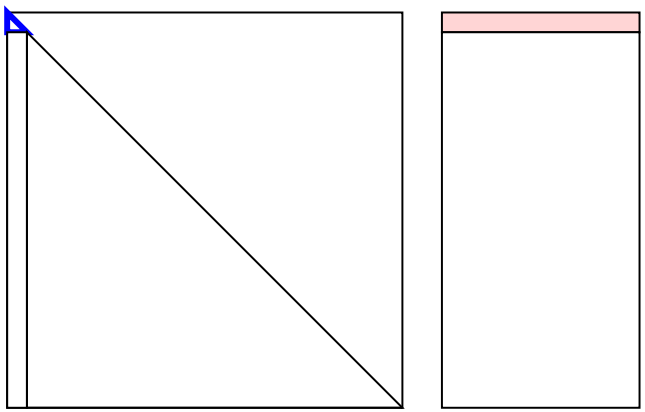
```
/* loop */
trsm(L, B) {
  for (k = 1; k ≤ M; k ++ ) {
    B_{k,1:M} /= l_{kk};
    B_{k+1:M,1:M} -= L_{k+1:M,k} B_{k,1:M};
  }
}
```



Recursions and loops

Again, partitioning is arbitrary and there is a loop-like partitioning

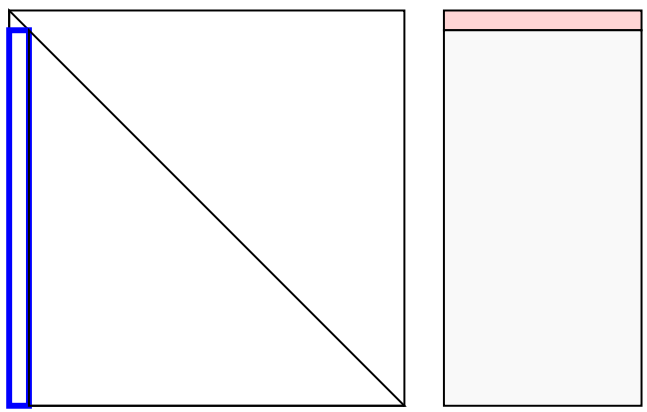
```
/* loop */
trsm(L, B) {
  for (k = 1; k ≤ M; k ++ ) {
    B_{k,1:M} /= l_{kk};
    B_{k+1:M,1:M} -= L_{k+1:M,k} B_{k,1:M};
  }
}
```



Recursions and loops

Again, partitioning is arbitrary and there is a loop-like partitioning

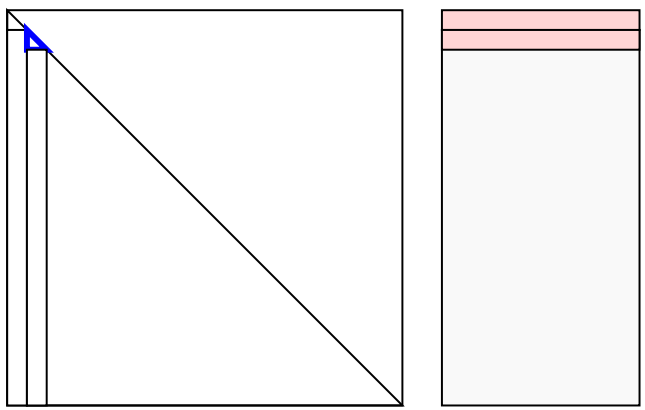
```
/* loop */
trsm(L, B) {
  for (k = 1; k ≤ M; k ++ ) {
    B_{k,1:M} /= l_{kk};
    B_{k+1:M,1:M} -= L_{k+1:M,k} B_{k,1:M};
  }
}
```



Recursions and loops

Again, partitioning is arbitrary and there is a loop-like partitioning

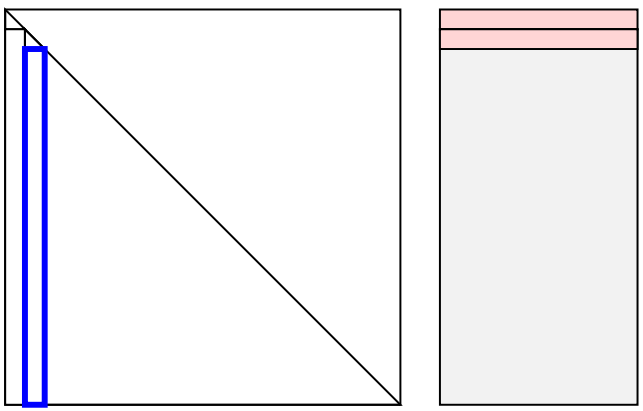
```
/* loop */
trsm(L, B) {
  for (k = 1; k ≤ M; k ++ ) {
    B_{k,1:M} /= l_{kk};
    B_{k+1:M,1:M} -= L_{k+1:M,k} B_{k,1:M};
  }
}
```



Recursions and loops

Again, partitioning is arbitrary and there is a loop-like partitioning

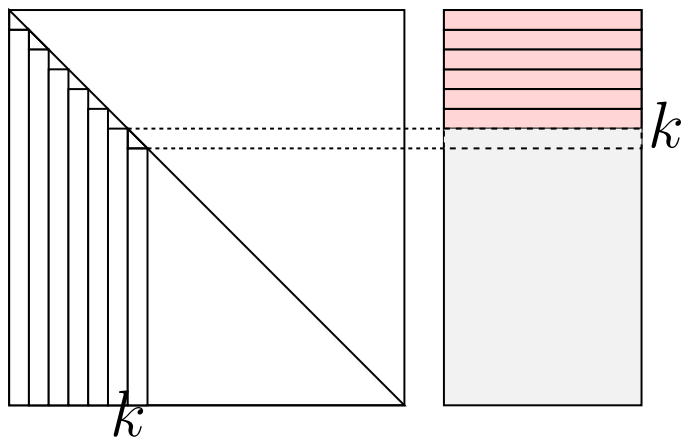
```
/* loop */
trsm(L, B) {
  for (k = 1; k ≤ M; k ++ ) {
    B_{k,1:M} /= l_{kk};
    B_{k+1:M,1:M} -= L_{k+1:M,k} B_{k,1:M};
  }
}
```



Recursions and loops

Again, partitioning is arbitrary and there is a loop-like partitioning

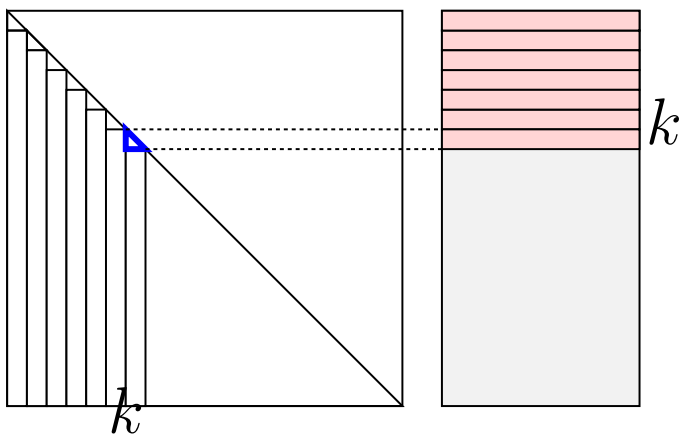
```
/* loop */
trsm(L, B) {
  for (k = 1; k ≤ M; k ++ ) {
    B_{k,1:M} /= l_{kk};
    B_{k+1:M,1:M} -= L_{k+1:M,k} B_{k,1:M};
  }
}
```



Recursions and loops

Again, partitioning is arbitrary and there is a loop-like partitioning

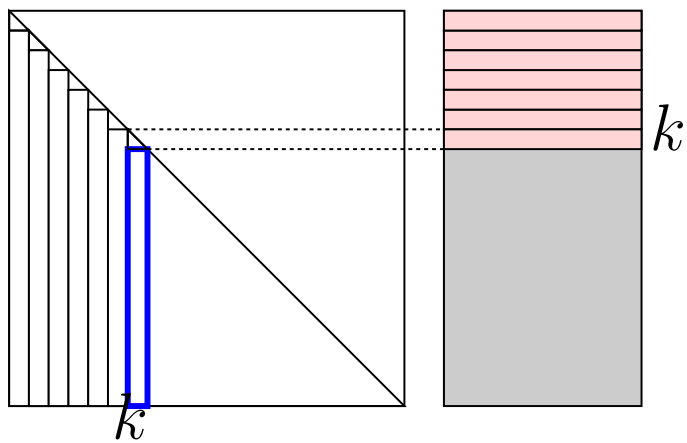
```
/* loop */
trsm(L, B) {
  for (k = 1; k ≤ M; k ++ ) {
    B_{k,1:M} /= l_{kk};
    B_{k+1:M,1:M} -= L_{k+1:M,k} B_{k,1:M};
  }
}
```



Recursions and loops

Again, partitioning is arbitrary and there is a loop-like partitioning

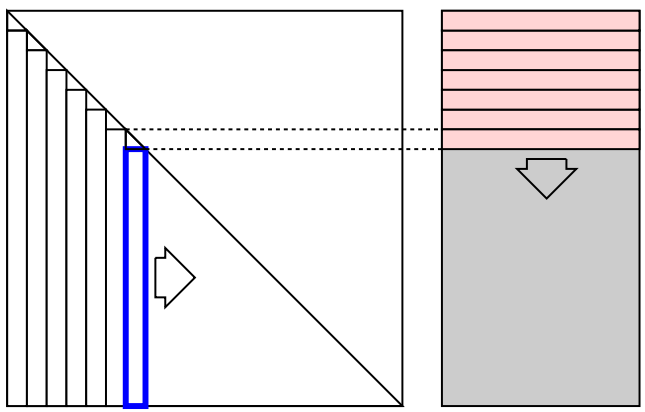
```
/* loop */
trsm(L, B) {
  for (k = 1; k ≤ M; k ++ ) {
    B_{k,1:M} /= l_{kk};
    B_{k+1:M,1:M} -= L_{k+1:M,k} B_{k,1:M};
  }
}
```



Recursions and loops

Again, partitioning is arbitrary and there is a loop-like partitioning

```
/* loop */
trsm(L, B) {
  for (k = 1; k ≤ M; k ++ ) {
    B_{k,1:M} /= l_{kk};
    B_{k+1:M,1:M} -= L_{k+1:M,k} B_{k,1:M};
  }
}
```

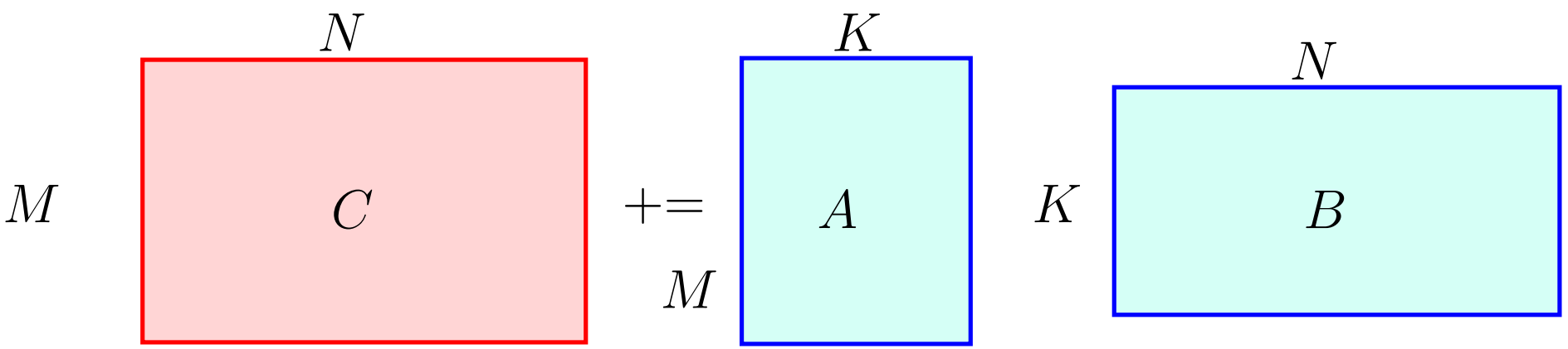


Contents

A subproblem 2: matrix multiply

- Input :
  - C: $M \times N$ matrix
  - A: $M \times K$ matrix
  - B: $K \times N$ matrix
- Output :

C += AB



Formulate using subproblems

Three ways to decompose

- divide M :

$$\begin{pmatrix} C_1 \\ C_2 \end{pmatrix} \texttt{ += } \begin{pmatrix} A_1 \\ A_2 \end{pmatrix} B$$

  $\rightarrow C_1 \texttt{ += } A_1 B \texttt{ // } C_2 \texttt{ += } A_2 B$

- divide N :

$$\begin{pmatrix} C_1 & C_2 \end{pmatrix} \texttt{ += } A \begin{pmatrix} B_1 & B_2 \end{pmatrix}$$

  $\rightarrow C_1 \texttt{ += } A B_1 \texttt{ // } C_2 \texttt{ += } A B_2$

- divide K :

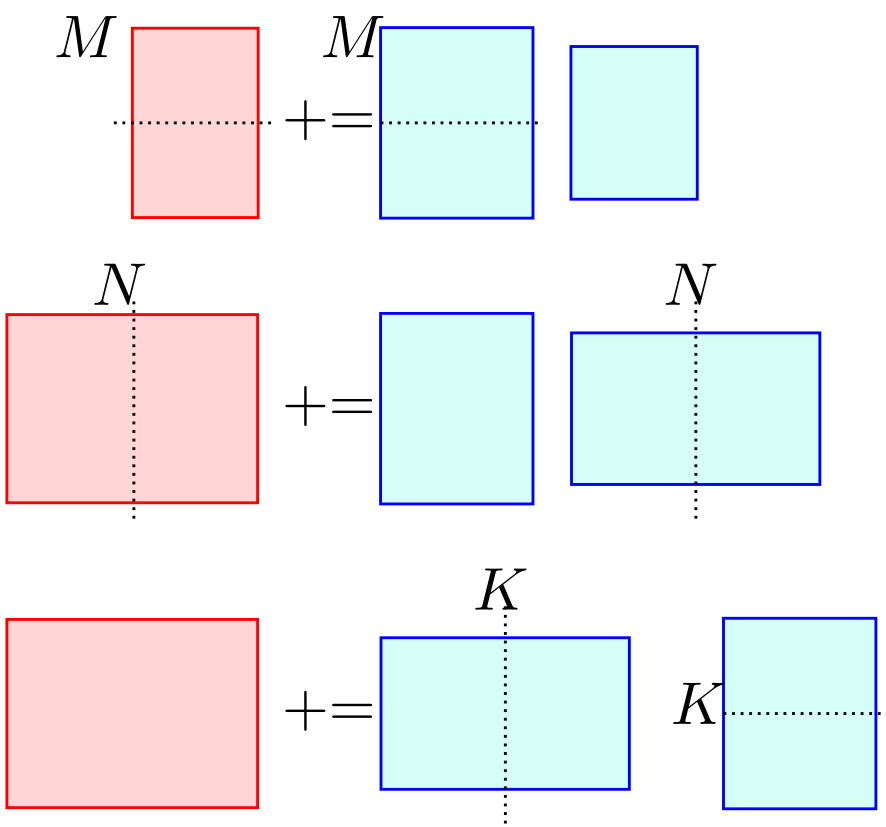
$$C \texttt{ += } \begin{pmatrix} A_1 & A_2 \end{pmatrix} \begin{pmatrix} B_1 \\ B_2 \end{pmatrix}$$

  $\rightarrow C \texttt{ += } A_1 B_1 \texttt{ ; } C \texttt{ += } A_2 B_2$

Which decomposition should we use?

- For reasons described later, divide the largest one among M, N, and K
- Make the shape of subproblems as square as possible

Solving using recursions




```
1  gemm(A, B, C) {
2    if ((M, N, K) = (1, 1, 1)) {
3      c11 += a11 * b11;
4    } else if (M ≥ N and M ≥ K) {
5      A1, A2 = split_h(A);
6      C1, C2 = split_h(C);
7      gemm(A1, B, C1);
8      gemm(A2, B, C2);
9    } else if (N ≥ M and N ≥ K)
10     B1, B2 = split_v(B);
11     C1, C2 = split_v(C);
12     gemm(A, B1, C1);
13     gemm(A, B1, C2);
14   } else {
15     A1, A2 = split_v(A);
16     B1, B2 = split_h(B);
17     gemm(A1, B1, C);
18     gemm(A2, B2, C);
19   }
20 }
```

Where is parallelism in our example? Cholesky

```
1  /* in place */
2  chol(A) {
3    if (n = 1) a11 := √a11;
4    else {
5      chol(A11);
6      /* triangular solve */
7      trsm(A11, A12);
8      A21 = tA12;
9      A22 -= A21A12
10     chol(A22);
11   }
12 }
```

- data dependency prohibits any of function calls in line 5-10 to be executed in parallel

Where is parallelism in our example? Triangular solve

```
1  /* triangular solve LX = B.
2     replace B with X */
3  trsm(L, B) {
4    if (M = 1) {
5      B /= l11;
6    } else if (M ≥ N) {
7      trsm(L11, B1);
8      B2 -= L21 B1;
9      trsm(L22, B2);
10   } else {
11     trsm(L, B1);
12     trsm(L, B2);
13   }
14 }
```

- function calls in line 7-9 cannot be run in parallel
- two calls to trsm at line 11 and a2 *can* be run in parallel

Where is parallelism in our example? Matrix multiply

```
gemm(A, B, C) {
  if ((M, N, K) = (1, 1, 1)) {
    c11 += a11 * b11;
  } else if (M ≥ N and M ≥ K) {
    A1, A2 = split_h(A);
    C1, C2 = split_h(C);
    gemm(A1, B, C1);
    gemm(A2, B, C2);
  } else if (N ≥ M and N ≥ K)
    B1, B2 = split_v(B);
    C1, C2 = split_v(C);
    gemm(A, B1, C1);
    gemm(A, B1, C2);
  } else {
    A1, A2 = split_v(A);
    B1, B2 = split_h(B);
    gemm(A1, B1, C);
    gemm(A2, B2, C);
  }
}
```

- when dividing M and N, two recursive calls can be parallel
- when dividing K, they should be serial
- (alternatively, we can execute them in parallel using two different regions for C and then add them)

That's basically it!

```
1  gemm(A, B, C) {
2    if ((M, N, K) = (1, 1, 1)) {
3      c11 += a11 * b11;
4    } else if (M ≥ N and M ≥ K) {
5      tbb::task_group tg;
6      A1, A2 = split_h(A);
7      C1, C2 = split_h(C);
8      tg.run([&] { gemm(A1, B, C1); });
9      gemm(A2, B, C2);
10     tg.wait();
11   } else if (N ≥ M and N ≥ K)
12     tbb::task_group tg;
13     B1, B2 = split_v(B);
14     C1, C2 = split_v(C);
15     tg.run([&] { gemm(A, B1, C1); });
16     gemm(A, B1, C2);
17     tg.wait();
18   } else {
19     // same as before
20     ...
21   }
22 }
```

```
1  /* triangular solve LX = B.
2     replace B with X */
3  trsm(L, B) {
4    if (M = 1) {
5      B /= l11;
6    } else if (M ≥ N) {
7      trsm(L11, B1);
8      B2 -= L21 B1;
9      trsm(L22, B2);
10   } else {
11     task_group tg;
12     tg.run([&] { trsm(L, B1); });
13     trsm(L, B2);
14     tg.wait();
15   }
16 }
```

T_1 and T_∞ of matrix multiply

```
gemm(A, B, C) {
  if ((M, N, K) = (1, 1, 1)) {
    c11 += a11 * b11;
  } else if (M ≥ N and M ≥ K) {
    ...
    tg.run([&] { gemm(A1, B, C1); });
    gemm(A2, B, C2);
    tg.wait();
  } else if (N ≥ M and N ≥ K)
    ...
    tg.run([&] { gemm(A, B1, C1); });
    gemm(A, B1, C2);
    tg.wait();
  } else {
    ...
    gemm(A1, B1, C);
    gemm(A2, B2, C);
  }
}
```

Work (T_1), written by $W_{\text{gemm}}(M, N, K) =$

$$\begin{cases} \Theta(1) & ((M, N, K) = (1, 1, 1)) \\ 2W_{\text{gemm}}(M/2, N, K) + \Theta(1) & (M \text{ is largest}) \\ 2W_{\text{gemm}}(M, N/2, K) + \Theta(1) & (N \text{ is largest}) \\ 2W_{\text{gemm}}(M, N, K/2) + \Theta(1) & (K \text{ is largest}) \end{cases}$$

$\Rightarrow \Theta(MNK)$

T_1 and T_∞ of matrix multiply

```
gemm(A, B, C) {
  if ((M, N, K) = (1, 1, 1)) {
    c11 += a11 * b11;
  } else if (M ≥ N and M ≥ K) {
    ...
    tg.run([&] { gemm(A1, B, C1); });
    gemm(A2, B, C2);
    tg.wait();
  } else if (N ≥ M and N ≥ K)
    ...
    tg.run([&] { gemm(A, B1, C1); });
    gemm(A, B1, C2);
    tg.wait();
  } else {
    ...
    gemm(A1, B1, C);
    gemm(A2, B2, C);
  }
}
```

Critical path (T_∞), written by $C_{\text{gemm}}(M, N, K) =$

$$\begin{cases} \Theta(1) & ((M, N, K) = (1, 1, 1)), \\ C_{\text{gemm}}(M/2, N, K) + \Theta(1) & (M \text{ is largest}) \\ C_{\text{gemm}}(M, N/2, K) + \Theta(1) & (N \text{ is largest}) \\ 2C_{\text{gemm}}(M, N, K/2) + \Theta(1) & (N \text{ is largest}) \end{cases}$$

$\Rightarrow \Theta(\log M + \log N + K)$ (we consider it as $\Theta(K)$ for brevity)

T_1 and T_∞ of triangular solve

```
/* triangular solve LX = B.
   replace B with X */
trsm(L, B) {
  if (M = 1) {
    B /= l11;
  } else if (M ≥ N) {
    trsm(L11, B1);
    B2 -= L21 B1;
    trsm(L22, B2);
  } else {
    task_group tg;
    tg.run([&] { trsm(L, B1); });
    trsm(L, B2);
    tg.wait();
  }
}
```

Work (T_1), written by

$$W_{\text{trsm}}(M,N) = \begin{cases} \Theta(1) & ((M,N)=(1,1,1)) \\ 2W_{\text{trsm}}(M/2,N) & \\ \quad +W_{\text{gemm}}(M/2,N,M/2) & (M \geq N) \\ 2W_{\text{trsm}}(M,N/2)+\Theta(1) & (N > M) \end{cases}$$

$$\Rightarrow \Theta(M^2 N)$$

T_1 and T_∞ of triangular solve

```
/* triangular solve LX = B.
   replace B with X */
trsm(L, B) {
  if (M = 1) {
    B /= l11;
  } else if (M ≥ N) {
    trsm(L11, B1);
    B2 -= L21 B1;
    trsm(L22, B2);
  } else {
    task_group tg;
    tg.run([&] { trsm(L, B1); });
    trsm(L, B2);
    tg.wait();
  }
}
```

Critical path (T_∞), written by $C_{\text{trsm}}(M, N) =$

$$\begin{cases} \Theta(1) & ((M,N) = (1,1)), \\ 2C_{\text{trsm}}(M/2, N) & \\ \quad + C_{\text{gemm}}(M/2, N, M/2) & (M \geq N) \\ C_{\text{trsm}}(M, N/2) + \Theta(1) & (N > M) \end{cases}$$

$\Rightarrow \Theta(M \log N)$

T_1 and T_∞ of Cholesky

```
chol(A) {
  if (n = 1) a11 := √a11;
  else {
    chol(A11);
    /* triangular solve */
    trsm(A11, A12);
    A21 = tA12;
    A22 -= A21A12
    chol(A22);
  }
}
```

Work (T_1), written by $W_{\mathrm{chol}}(n) =$

$$\begin{cases} \Theta(1) & (n=1), \\ 2W_{\mathrm{chol}}(n/2) \\ \quad +W_{\mathrm{trsm}}(n/2, n/2) \\ \quad +W_{\mathrm{trans}}(n/2, n/2) \\ \quad +W_{\mathrm{gemm}}(n/2, n/2, n/2) \end{cases}$$

$\Rightarrow \Theta(n^3)$

T_1 and T_∞ of Cholesky

```
chol(A) {
  if (n = 1) a11 := √a11;
  else {
    chol(A11);
    /* triangular solve */
    trsm(A11, A12);
    A21 = tA12;
    A22 -= A21A12
    chol(A22);
  }
}
```

Critical path (T_∞), written by $C_{\text{chol}}(n) =$

$$\begin{cases} \Theta(1) & (n = 1) \\ 2C_{\text{chol}}(n/2) & \\ \quad +C_{\text{trsm}}(n/2, n/2) & \\ \quad +C_{\text{trans}}(n/2, n/2) & \\ \quad +C_{\text{gemm}}(n/2, n/2, n/2) & \end{cases}$$

$\Rightarrow \Theta(n \log n)$

Summary

For $n \times n$ matrix,

- $T_1 \in \Theta(n^3)$
- $T_\infty \in \Theta(n \log n)$
- the average parallelism:

$$T_1/T_\infty = \frac{n^2}{\log n}$$

- this should be ample for sufficiently large n
- a constant thresholding does not affect the asymptotic result;
  - you can switch to a serial loop for matrices smaller than a constant
- in practice, this threshold affects T_1 and T_∞
  - T_1 will decrease (good thing)
  - T_∞ will increase due to a larger serial computation at leaves

Hands-on exercise next week

- please bring your laptop with SSH
- details on the necessary preparation are (hopefully) on the home page until the weekend