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

Kenjiro Taura

# Contents

# Contents

# Goals

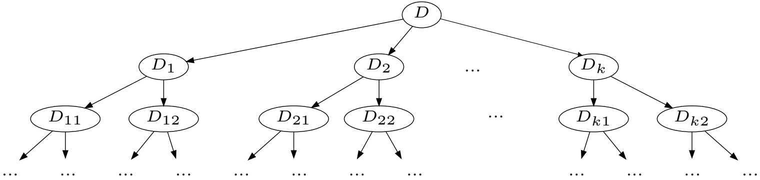
learn:

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

# Divide and conquer algorithms

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

```
answer solve(D) {
  if ( trivial (D)) {
    return trivially_solve (D);
  } else {
    D_1, ..., D_k = divide(D); // divide the problem into sub problems
    a_1 = solve(D_1); ...; a_k = solve(D_k); // solve them
    return combine(a_1, ..., a_k); // combine sub answers
  }
}
```



# 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

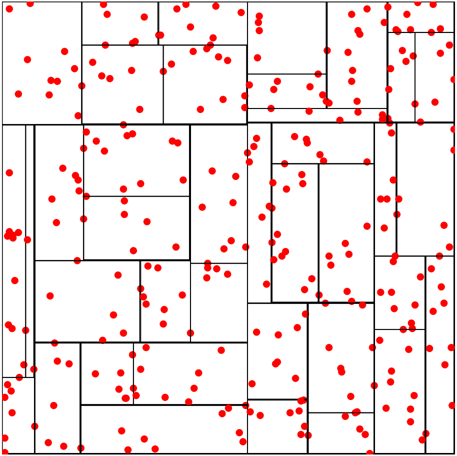
- quick sort, merge sort
- matrix multiply, LU factorization, eigenvalue
- FFT, polynomial multiply, big int multiply
- maximum segment sum, find median
- $k$-d tree
- ...

# Contents

# Contents

# $k$-d tree

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



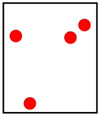
# $k$-d tree construction

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

# $k$-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 $t$ must satisfy
  1. each leaf has $\leq c$ points

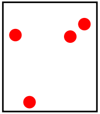
Leaf:



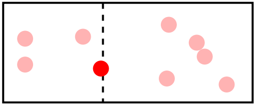
# $k$-d tree construction

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

Leaf:



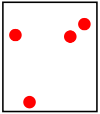
Internal:



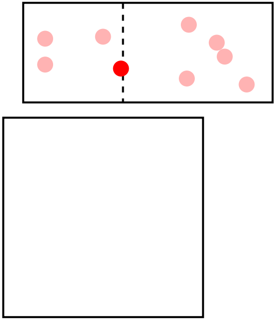
# $k$-d tree construction

- Input:
  - $P$: an array of points (no particular order)
  - $R$: a bounding box of $P$
- Output:
  - $t$: a $k$-d tree for $P$
- Properties $t$ must satisfy
  1. each leaf has $\leq c$ points
  2. each internal node has one point of its own plus one or two children
  3. each internal node is split into two subspaces by a line passing through its point
  4. axis to split a node perpendicular to is chosen alternately (first the $x$-axis, then the $y$-axis, and so on)

Leaf:

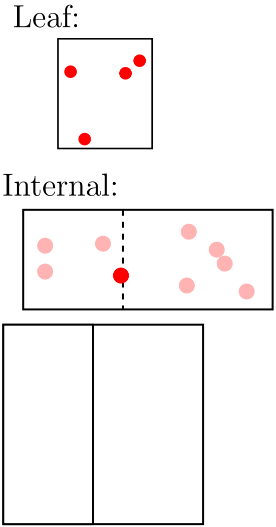


Internal:



# $k$-d tree construction

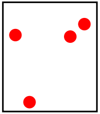
- Input:
  - $P$: an array of points (no particular order)
  - $R$: a bounding box of $P$
- Output:
  - $t$: a $k$-d tree for $P$
- Properties $t$ must satisfy
  1. each leaf has $\leq c$ points
  2. each internal node has one point of its own plus one or two children
  3. each internal node is split into two subspaces by a line passing through its point
  4. axis to split a node perpendicular to is chosen alternately (first the $x$-axis, then the $y$-axis, and so on)



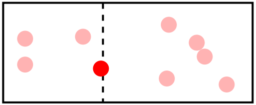
# $k$-d tree construction

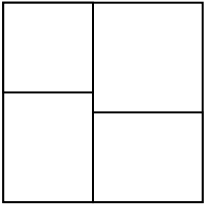
- Input:
  - $P$: an array of points (no particular order)
  - $R$: a bounding box of $P$
- Output:
  - $t$: a $k$-d tree for $P$
- Properties $t$ must satisfy
  1. each leaf has $\leq c$ points
  2. each internal node has one point of its own plus one or two children
  3. each internal node is split into two subspaces by a line passing through its point
  4. axis to split a node perpendicular to is chosen alternately (first the $x$-axis, then the $y$-axis, and so on)

Leaf:



Internal:

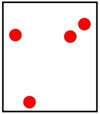




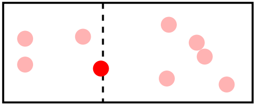
# $k$-d tree construction

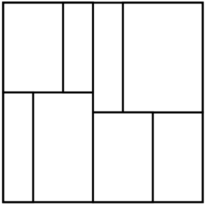
- Input:
  - $P$: an array of points (no particular order)
  - $R$: a bounding box of $P$
- Output:
  - $t$: a $k$-d tree for $P$
- Properties $t$ must satisfy
  1. each leaf has $\leq c$ points
  2. each internal node has one point of its own plus one or two children
  3. each internal node is split into two subspaces by a line passing through its point
  4. axis to split a node perpendicular to is chosen alternately (first the $x$-axis, then the $y$-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

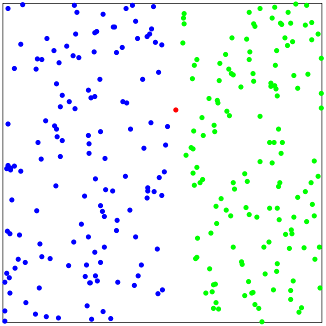
# How to build a $k$-d tree

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

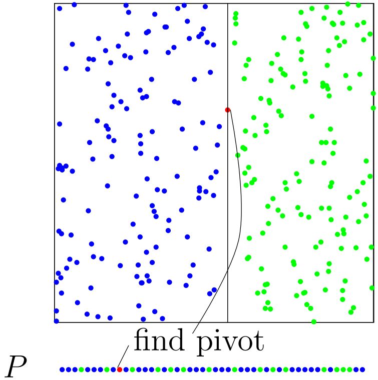
# divide and conquer method

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



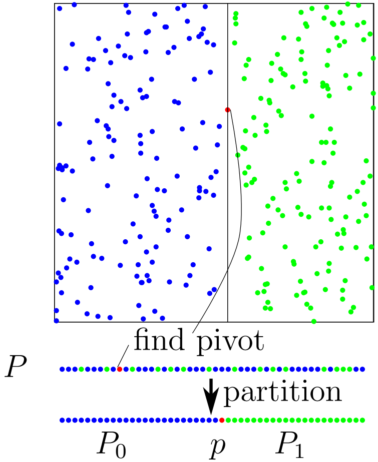
# divide and conquer method

- to build a tree for a rectangle $R$ and points $P$ in $R$,
- choose a “pivot” $p \in P$ 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 "pivot" $p \in P$ through which to split $R$, and
- partition $P$ into $P_0 + \{p\} + P_1$ where,
  - if we split perpendicular to the $x$-axis,
  - $P_0$ : points whose $x$ coodinate $<$ $p$'s
  - $P_1$ : points whose $x$ coodinate $\geq$ $p$'s (except $p$)



# divide and conquer method

```
/* build a k-d tree for a set of points P in a rectangular region R and return
   the root of the tree. the node is at depth, so it should split along
   (depth % D)th axis */
build(P, R, depth) {
  if (|P| == 0) {
    return 0; /* empty */
  } else if (|P| <= threshold) {
    /* small enough; leaf */
    return make_leaf(P, R, depth);
  } else {
    /* find a point whose coordinate to split is near the median */
    p = find_pivot(P, depth % D);
    /* split R into two sub-rectangles */
    R0,R1 = split_rect(R, depth % D, p.pos[depth % D]);
    /* partition P by their coodinate lower/higher than p's coordinate */
    P0,P1 = partition(P - { p }, depth % D, p.pos[depth % D]);
    /* build a tree for each rectangle */
    n0 = build(P0, R0, depth + 1);
    n1 = build(P1, R1, depth + 1);
    /* return a node having n0 and n1 as its children */
    return make_node(p, n0, n1, depth);
  }
}
```

# Notes on subprocedures

- $p =$ `find_pivot`$(P,\, d)$
  - find a point $\in P$ whose $d$th 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 "$d$th 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 $P_0$ ($d$th coordinate $< c$) and $P_1$ ($d$th coordinate $\geq c$)
  - $\Rightarrow O(|P|)$

# 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 (`#pragma omp parallel`, `master`, `task`, `taskwait`)
  - Intel Threading Building Block (TBB)
  - Cilk, CilkPlus

# Parallelizing *k*-d tree construction with tasks

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

```
build(P, R, depth) {
  if (|P| == 0) {
    return 0; /* empty */
  } else if (|P| <= threshold) {
    return make_leaf(P, R, depth);
  } else {
    p = find_pivot(P, depth % D);
    R0,R1 = split_rect(R, depth % D, p.pos[depth % D]);
    P0,P1 = partition(P - { p }, depth % D, p.pos[depth % D]);
#pragma omp task shared(n0)
    n0 = build(P0, R0, depth + 1);
#pragma omp task shared(n1)
    n1 = build(P1, R1, depth + 1);
#pragma omp taskwait
    return make_node(p, n0, n1, depth);
  }
}
```

- cumbersome to parallelize with only parallel loops

# 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

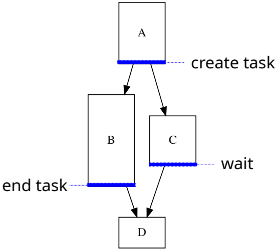
- **Work:** = the total amount of work of the computation
  - = the time it takes in a serial execution
- **Critical path length:** = the maximum length of dependent chain of computation
  - a more precise definition follows, with *computational DAGs*

# Computational DAGs

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

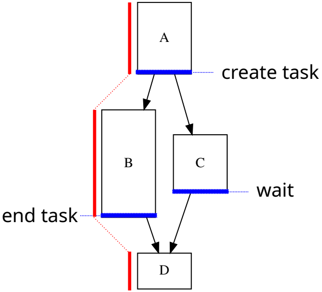
- a node = an interval of computation free of task parallel primitives
  - i.e. a node *starts* and *ends* by a task parallel primitive
  - we assume a single node is executed non-preemptively
- an edge = a dependency between two nodes, of three types:
  - parent → created child
  - child → waiting parent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

```
main() {
  A();
  create_task B();
  C();
  wait(); // wait for B
  D();
}
```



# 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



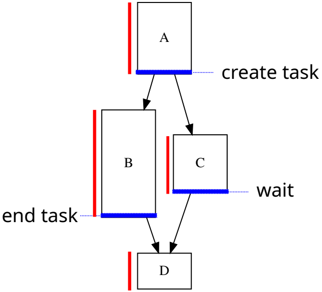

Given a computational DAG,

> *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



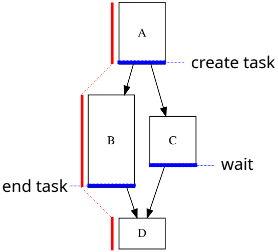

Given a computational DAG,

$$\textit{work} = \textit{the sum of lengths of all nodes}$$

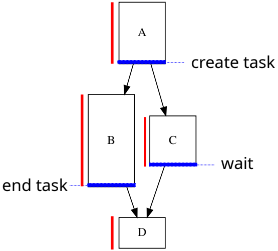
# What do they intuitively mean?

- 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 $T_\infty$
- Analogously, we often denote *work* by $T_1$

$T_1 = $ *work*, $T_\infty = $ *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 $T_\infty$,

  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 $T_\infty$,

  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!

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

# A few facts to remember about $T_1$ and $T_\infty$

Consider the execution time with $P$ processors ($T_P$)

- there are two obvious *lower bounds*
  - $T_P \geq \frac{T_1}{P}$
  - $T_P \geq T_\infty$

  or more simply,

$$T_P \geq \max(\frac{T_1}{P}, T_\infty)$$

- what a greedy scheduler achieves is

$$T_P \leq \text{sum}(\frac{T_1}{P}, T_\infty)$$

- two memorable facts
  - *"the sum of two lower bounds is an upper bound"*
  - any greedy scheduler is within a factor of two of the optimal scheduler (下手な考え休むに似たり?)

# A few facts to remember about $T_1$ and $T_\infty$

- to get good (nearly perfect) speedup, we wish to have

$$\frac{T_1}{P} \gg T_\infty$$

  or equivalently,

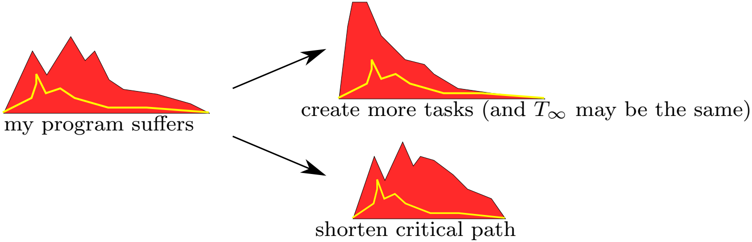
$$\frac{T_1}{T_\infty} \gg P$$

- we can consider $\frac{T_1}{T_\infty}$ to be *the average parallelism* (the speedup we would get with infinitely many processors)
- we like to make the average parallelism large enough compared to the actual number of processors

# Takeaway message

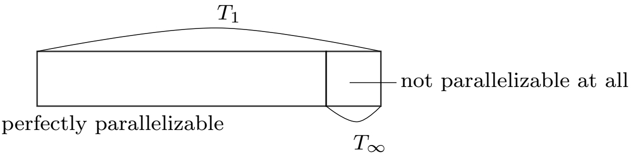
*Suffer from low speedup?* $\Rightarrow$ *try to shorten the critical path*

*people are tempted to think creating more and more tasks is the way; they are useless, if it does not shorten the critical path*



# A special case (1) — Amdahl's law

- assume the entire computation ($T_1$) consists of two parts,
  1. one completely serial ($T_\infty$), and
  2. the other completely parallelizable ($T_1 - T_\infty$)



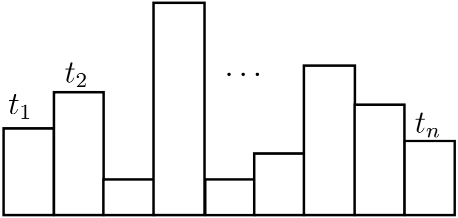

- Amdahl's law states $T_p \geq T_\infty$, which is trivial
- it is also trivial to observe that *any* greedy scheduler achieves

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

  which coincides with what the greedy scheduler theorem says (for more general cases)
- takeaway: want to get a good speedup? $\Rightarrow$ minimize $T_\infty$, or the work not parallelized

# A special case (2) — "bag of tasks"

- assume we have a set of $n$ indepent (serial) tasks whose runtimes are $t_1, t_2, \cdots, t_n$




- consider a dynamic greedy scheduler in which each core repeats fetching a task at a time and executing it
- then
  - $T_1 = t_1 + t_2 + \cdots + t_n$
  - $T_\infty = \max(t_1, t_2, \cdots, t_n)$
- takeaway : you want to get a good speedup? $\Rightarrow$ shorten $\max(t_1, t_2, \cdots, t_n)$, or the execution time of the *longest* task

# What makes $T_\infty$ so useful?

$T_\infty$ 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)

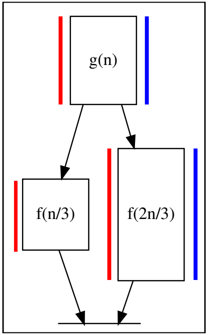
# Contents

# Calculating work and critical path

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

```
f(n) {
  if (n == 1) { trivial(n); /* assume O(1) */ }
  else {
    g(n);
    create_task f(n/3);
    f(2*n/3);
    wait();
  }
}
```

f(n) =




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

```
build(P, R, depth) {
  if (|P| == 0) {
    return 0; /* empty */
  } else if (|P| <= threshold) {
    return make_leaf(P, R, depth);
  } else {
    p = find_pivot(P, depth % D);
    R0,R1 = split_rect(R, depth % D, p.pos[depth % D]);
    P0,P1 = partition(P - { p }, depth % D, p.pos[depth % D]);
    n0 = create_task build(P0, R0, depth + 1);
    n1 = build(P1, R1, depth + 1);
    wait();
    return make_node(p, n0, n1, depth);
  } }
```

recall that `partition` takes time proportional to $n$ (the number of points). thus,

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

omitting math,

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

# Remark

- the argument above is crude and optimistic, as $n$ points are not always split into two sets of equal sizes
- omitting math again, 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

```
build(P, R, depth) {
  if (|P| == 0) {
    return 0; /* empty */
  } else if (|P| <= threshold) {
    return make_leaf(P, R, depth);
  } else {
    p = find_pivot(P, depth % D);
    R0,R1 = split_rect(R, depth % D, p.pos[depth % D]);
    P0,P1 = partition(P - { p }, depth % D, p.pos[depth % D]);
    n0 = create_task build(P0, R0, depth + 1);
    n1 = build(P1, R1, depth + 1);
    wait();
    return make_node(p, n0, n1, depth);
  } }
```

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

omitting math,

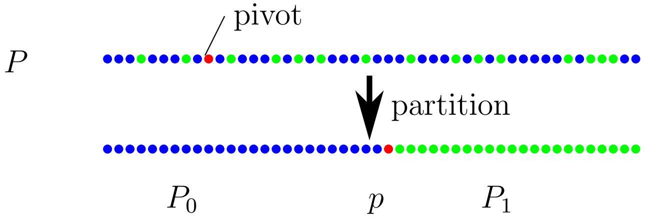
$$\therefore C_{\text{build}}(n) \in \Theta(n)$$

# 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`



# Contents

# 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

```
/* sort a..a_end and put the result into
   (i)  a (if dest = 0)
   (ii) t (if dest = 1) */
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 */
    ms(a, c,     t,      1 - dest);
    /* sort 2nd half */
    ms(c, a_end, t + nh, 1 - dest);
    elem * s = (dest ? a : t);
    elem * d = (dest ? t : a);
    /* merge them */
    merge(s,      s + nh,
          s + nh, s + n, d);
  }
}
```

```
/* 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++;
}
```

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);
  }
}
```

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

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

# 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);
  }
}
```

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

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

# Speedup of merge sort

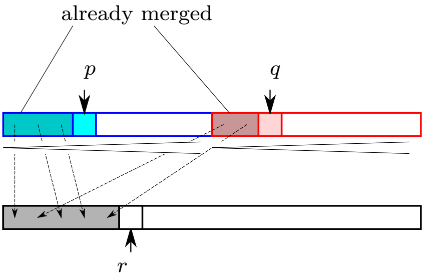
$$T_1 = W_{\mathrm{ms}}(n) \in \Theta(n \log n),$$
$$T_\infty = C_{\mathrm{ms}}(n) \in \Theta(n).$$

the average parallelism

$$T_1/T_\infty \in \Theta(\log n).$$

# How (serial) merge works

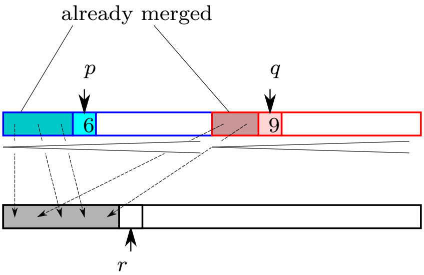
```
/* merge a_beg ... a_end
     and b_beg ... b_end
    into c */
void
merge(elem * a, elem * a_end,
      elem * b, elem * b_end,
      elem * c) {
  elem * p = a, * q = b, * r = c;
  while (p < a_end && q < b_end) {
    if (*p < *q) { *r++ = *p++; }
    else { *r++ = *q++; }
  }
  while (p < a_end)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




Looks very serial ...

# How (serial) merge works

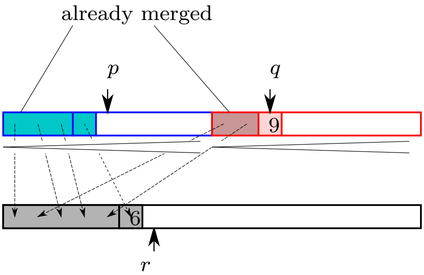
```
/* merge a_beg ... a_end
     and b_beg ... b_end
    into c */
void
merge(elem * a, elem * a_end,
      elem * b, elem * b_end,
      elem * c) {
  elem * p = a, * q = b, * r = c;
  while (p < a_end && q < b_end) {
    if (*p < *q) { *r++ = *p++; }
    else { *r++ = *q++; }
  }
  while (p < a_end)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

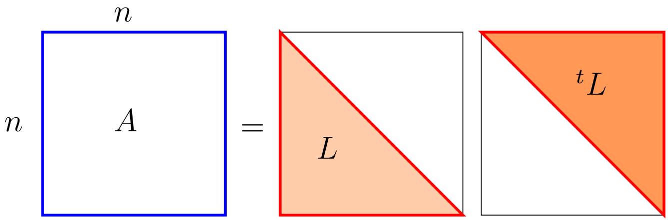
# Contents

# 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

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

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

```
/* Cholesky factorization */
chol(A) {
  if (n = 1) return (√a11);
  else {
    L11 = chol(A11);
    /* triangular solve */
    tL21 = trsm(L11, tA21);
    L22 = chol(A22 − L21 tL21);
    return ( L11  tL21 ; L21  L22 )
  }
}
```

# 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. $A_{22} = L_{21}{}^tL_{21} + L_{22}\,{}^tL_{22}$

*/* Cholesky factorization */*<br>
chol($A$) {<br>
  **if** $(n = 1)$ **return** $(\sqrt{a_{11}})$;<br>
  **else** {<br>
    $L_{11} = \mathrm{chol}(A_{11})$;<br>
    */* triangular solve */*<br>
    ${}^tL_{21} = \mathrm{trsm}(L_{11}, {}^tA_{21})$;<br>
    $L_{22} = \mathrm{chol}(A_{22} - L_{21}{}^tL_{21})$;<br>
    **return** $\begin{pmatrix} L_{11} & {}^tL_{21} \\ L_{21} & L_{22} \end{pmatrix}$<br>
  }<br>
}

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

$/*$ *Cholesky factorization* $*/$<br>
chol$(A)$ {<br>
  **if** $(n = 1)$ **return** $(\sqrt{a_{11}})$;<br>
  **else** {<br>
    $L_{11} = \mathrm{chol}(A_{11})$;<br>
    $/*$ *triangular solve* $*/$<br>
    ${}^tL_{21} = \mathrm{trsm}(L_{11}, {}^tA_{21})$;<br>
    $L_{22} = \mathrm{chol}(A_{22} - L_{21}\,{}^tL_{21})$;<br>
    **return** $\begin{pmatrix} L_{11} & {}^tL_{21} \\ L_{21} & L_{22} \end{pmatrix}$<br>
  }<br>
}

# 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 $L_{22}$

```
/* Cholesky factorization */
chol(A) {
  if (n = 1) return (√a11);
  else {
    L11 = chol(A11);
    /* triangular solve */
    tL21 = trsm(L11, tA21);
    L22 = chol(A22 − L21 tL21);
    return ( L11  tL21
             L21  L22 )
  }
}
```

$$\textbf{return}\ \begin{pmatrix} L_{11} & {}^tL_{21} \\ L_{21} & L_{22} \end{pmatrix}$$

# 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 */*
chol($A$) {
  **if** $(n = 1)$ **return** $(\sqrt{a_{11}})$;
  **else** {
    $L_{11} = \mathrm{chol}(A_{11})$;
    */* triangular solve */*
    ${}^tL_{21} = \mathrm{trsm}(L_{11}, {}^tA_{21})$;
    $L_{22} = \mathrm{chol}(A_{22} - L_{21}{}^tL_{21})$;
    **return** $\begin{pmatrix} L_{11} & {}^tL_{21} \\ L_{21} & L_{22} \end{pmatrix}$
  }
}

*/* in place */*
chol($A$) {
  **if** $(n = 1)$ $a_{11} := \sqrt{a_{11}}$;
  **else** {
    chol($A_{11}$);
    */* triangular solve */*
    trsm($A_{11}, A_{12}$);
    $A_{21} = {}^tA_{12}$;
    $A_{22}$ -= $A_{21}A_{12}$
    chol($A_{22}$);
  }
}

# In-place Cholesky at work

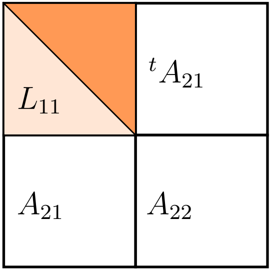
*/* in place */*
chol($A$) {
  **if** $(n = 1)\ a_{11} := \sqrt{a_{11}}$;
  **else** {
    chol($A_{11}$);
    */* triangular solve */*
    trsm($A_{11}, A_{12}$);
    $A_{21} = {}^{t}A_{12}$;
    $A_{22}$ **-=** $A_{21}A_{12}$
    chol($A_{22}$);
  }
}

$$\begin{array}{|c|c|} \hline A_{11} & {}^{t}A_{21} \\ \hline A_{21} & A_{22} \\ \hline \end{array}$$

# In-place Cholesky at work

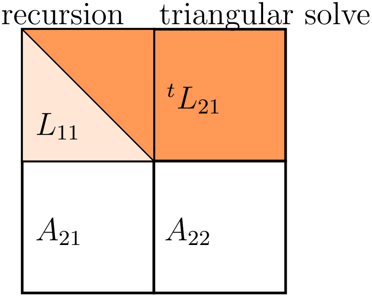
```
/* in place */
chol(A) {
  if (n = 1) a_11 := sqrt(a_11);
  else {
    chol(A_11);
    /* triangular solve */
    trsm(A_11, A_12);
    A_21 = tA_12;
    A_22 -= A_21 A_12
    chol(A_22);
  }
}
```

recursion



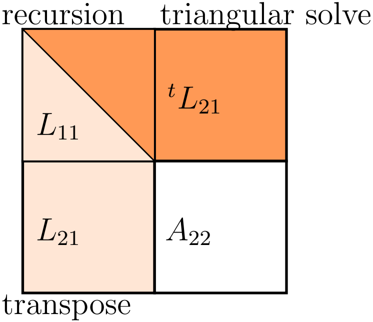
# In-place Cholesky at work

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



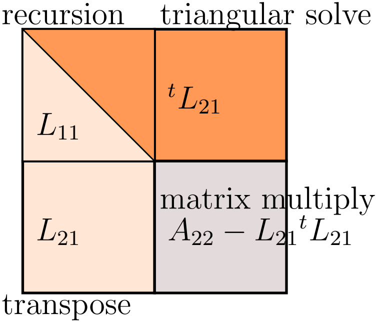
# In-place Cholesky at work

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



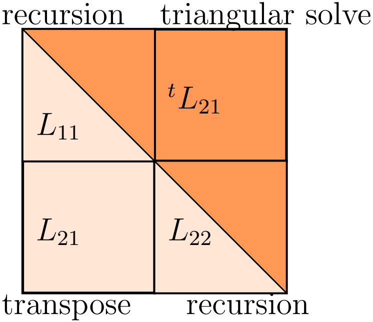
# In-place Cholesky at work

```
/* in place */
chol(A) {
  if (n = 1) a_11 := √a_11;
  else {
    chol(A_11);
    /* triangular solve */
    trsm(A_11, A_12);
    A_21 = ᵗA_12;
    A_22 -= A_21 A_12
    chol(A_22);
  }
}
```



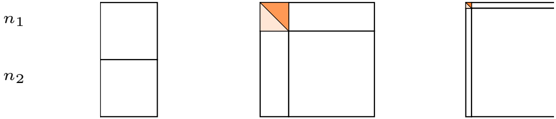
# In-place Cholesky at work

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



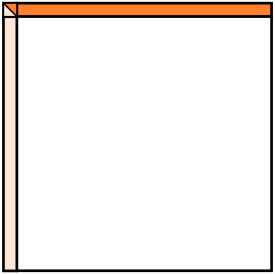
# Remark 2 : where to decompose

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



# Recursion to loops

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

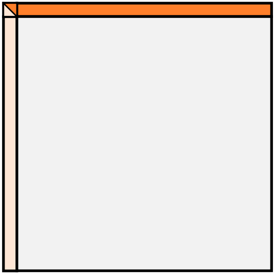


```
/* loop version */
chol_loop(A) {
  for (k = 1; k ≤ n; k ++ ) {
    a_{kk} := √a_{kk};
    A_{k,k+1:n} /= a_{kk};
    A_{k+1:n,k} /= a_{kk};
    A_{k+1:n,k+1:n} -= A_{k:n,k} A_{k,k:n}
  }
}
```

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

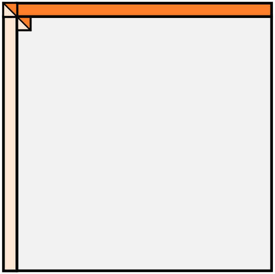


```
/* loop version */
chol_loop(A) {
  for (k = 1; k <= n; k ++ ) {
    a_kk := sqrt(a_kk);
    A_{k,k+1:n} /= a_kk;
    A_{k+1:n,k} /= a_kk;
    A_{k+1:n,k+1:n} -= A_{k:n,k} A_{k,k:n}
  }
}
```

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

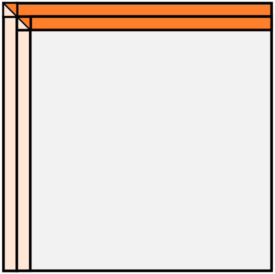


```
/* loop version */
chol_loop(A) {
  for (k = 1; k ≤ n; k ++ ) {
    a_kk := sqrt(a_kk);
    A_{k,k+1:n} /= a_kk;
    A_{k+1:n,k} /= a_kk;
    A_{k+1:n,k+1:n} -= A_{k:n,k} A_{k,k:n}
  }
}
```

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

# Recursion to loops

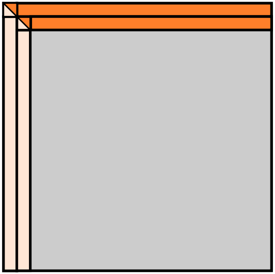
- The “loop-like” version (partition into $1 + (n-1)$) can be written in a true loop



```
/* loop version */
chol_loop(A) {
  for (k = 1; k ≤ n; k ++ ) {
    a_{kk} := √a_{kk};
    A_{k,k+1:n} /= a_{kk};
    A_{k+1:n,k} /= a_{kk};
    A_{k+1:n,k+1:n} -= A_{k:n,k} A_{k,k:n}
  }
}
```

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

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

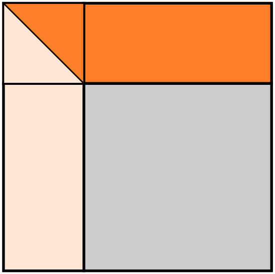


```
/* loop version */
chol_loop(A) {
  for (k = 1; k ≤ n; k ++ ) {
    a_kk := √a_kk;
    A_{k,k+1:n} /= a_kk;
    A_{k+1:n,k} /= a_kk;
    A_{k+1:n,k+1:n} -= A_{k:n,k} A_{k,k:n}
  }
}
```

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

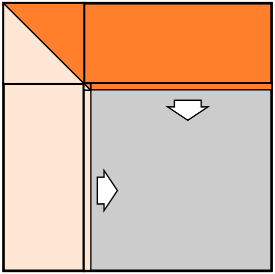


```
/* loop version */
chol_loop(A) {
  for (k = 1; k ≤ n; k ++ ) {
    a_{kk} := sqrt(a_{kk});
    A_{k,k+1:n} /= a_{kk};
    A_{k+1:n,k} /= a_{kk};
    A_{k+1:n,k+1:n} -= A_{k:n,k} A_{k,k:n}
  }
}
```

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



*/* loop version */*<br>
chol_loop($A$) {<br>
  **for** ($k = 1$; $k \leq n$; $k$ **++** ) {<br>
    $a_{kk} := \sqrt{a_{kk}}$;<br>
    $A_{k,k+1:n}$ **/=** $a_{kk}$;<br>
    $A_{k+1:n,k}$ **/=** $a_{kk}$;<br>
    $A_{k+1:n,k+1:n}$ **-=** $A_{k:n,k} A_{k,k:n}$<br>
  }<br>
}

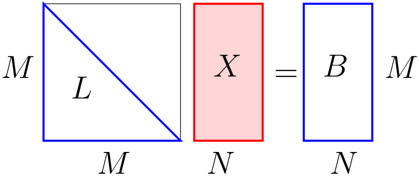
In practice, you still need to code the loop to avoid creating too small tasks

# 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.

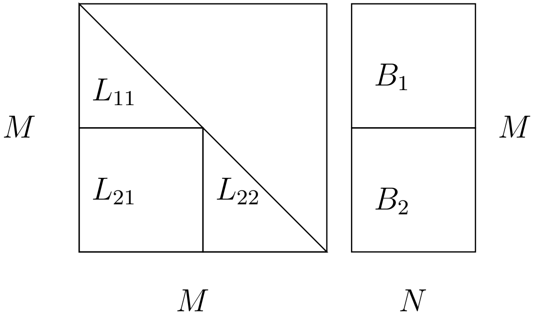
$$L \begin{pmatrix} X_1 & X_2 \end{pmatrix} = \begin{pmatrix} B_1 & B_2 \end{pmatrix} \Rightarrow$$

solve them independently (easy)

```
/* triangular solve LX = B.
   replace B with X */
trsm(L, B) {
  if (M = 1) {
    B /= l_11;
  } else if (M ≥ N) {
    trsm(L_11, B_1);
    B_2 -= L_21 B_1;
    trsm(L_22, B_2);
  } else {
    trsm(L, B_1);
    trsm(L, B_2);
  }
}
```

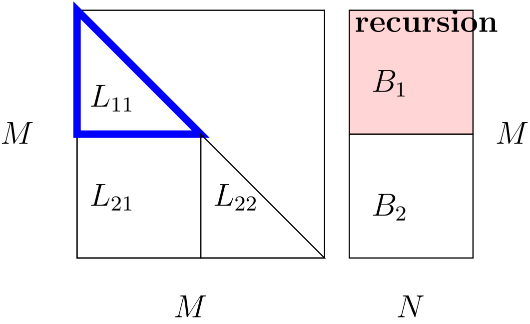
# Triangular solve at work

```
/* triangular solve LX = B.
   replace B with X */
trsm(L, B) {
  if (M = 1) {
    B /= l_11;
  } else if (M ≥ N) {
    trsm(L_11, B_1);
    B_2 -= L_21 B_1;
    trsm(L_22, B_2);
  } else {
    trsm(L, B_1);
    trsm(L, B_2);
  }
}
```



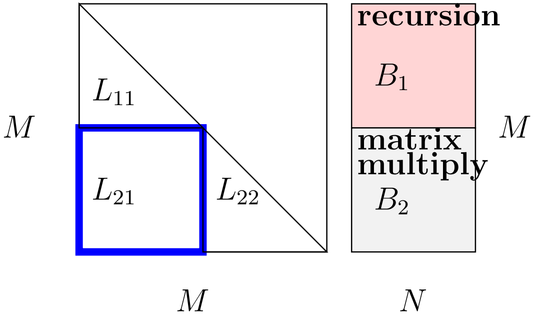
# Triangular solve at work

```
/* 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);
  }
}
```



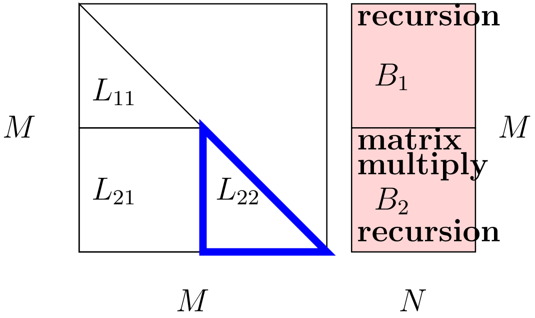
# Triangular solve at work

```
/* 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

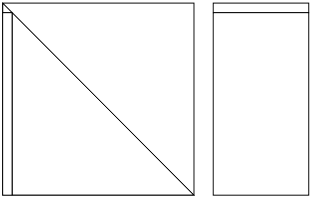
```
/* triangular solve LX = B.
   replace B with X */
trsm(L, B) {
  if (M = 1) {
    B /= l11;
  } else if (M ≥ N) {
    trsm(L11, B1);
    B2 -= L21 B1;
    trsm(L22, B2);
  } else {
    trsm(L, B1);
    trsm(L, B2);
  }
}
```



# Recursions and loops

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

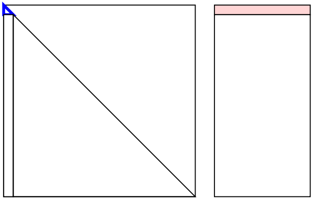
```
/* loop */
trsm(L, B) {
  for (k = 1; k ≤ M; k ++ ) {
    B_{k,1:M} /= l_{kk};
    B_{k+1:M,1:M} -= L_{k+1:M,k} B_{k,1:M};
  }
}
```



# Recursions and loops

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

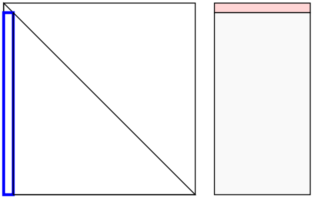
$/*$ *loop* $*/$
trsm$(L, B)$ {
  **for** $(k = 1;\ k \leq M;\ k$ ++ ) {
    $B_{k,1:M}$ /= $l_{kk}$;
    $B_{k+1:M,1:M}$ -= $L_{k+1:M,k} B_{k,1:M}$;
  }
}



# Recursions and loops

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

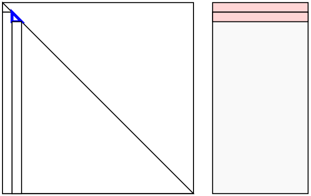
```
/* loop */
trsm(L, B) {
  for (k = 1; k ≤ M; k ++ ) {
    B_{k,1:M} /= l_{kk};
    B_{k+1:M,1:M} -= L_{k+1:M,k} B_{k,1:M};
  }
}
```



# Recursions and loops

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

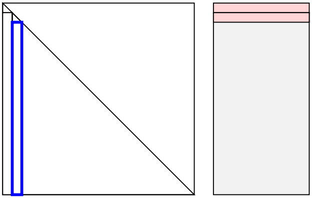
```
/* loop */
trsm(L, B) {
  for (k = 1; k ≤ M; k ++ ) {
    B_{k,1:M} /= l_{kk};
    B_{k+1:M,1:M} -= L_{k+1:M,k} B_{k,1:M};
  }
}
```



# Recursions and loops

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

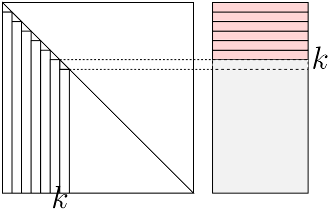
/* *loop* */
trsm($L, B$) {
  **for** ($k = 1$; $k \leq M$; $k$ **++** ) {
    $B_{k,1:M}$ **/=** $l_{kk}$;
    $B_{k+1:M,1:M}$ **-=** $L_{k+1:M,k} B_{k,1:M}$;
  }
}



# Recursions and loops

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

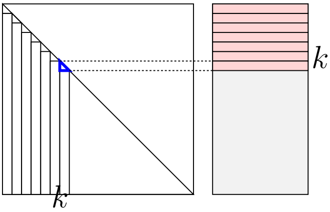
```
/* loop */
trsm(L, B) {
  for (k = 1; k ≤ M; k ++ ) {
    B_{k,1:M} /= l_{kk};
    B_{k+1:M,1:M} -= L_{k+1:M,k} B_{k,1:M};
  }
}
```



# Recursions and loops

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

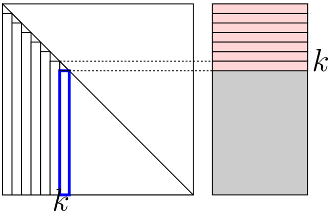
```
/* loop */
trsm(L, B) {
  for (k = 1; k ≤ M; k ++ ) {
    B_{k,1:M} /= l_{kk};
    B_{k+1:M,1:M} -= L_{k+1:M,k} B_{k,1:M};
  }
}
```



# Recursions and loops

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

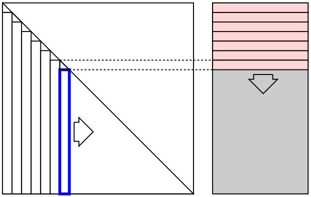
```
/* loop */
trsm(L, B) {
  for (k = 1; k ≤ M; k ++ ) {
    B_{k,1:M} /= l_{kk};
    B_{k+1:M,1:M} -= L_{k+1:M,k} B_{k,1:M};
  }
}
```



# 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};
  }
}
```

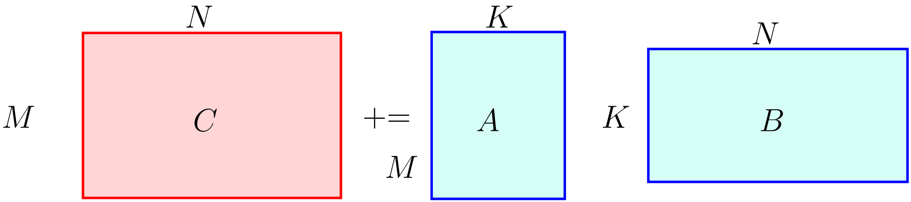


# Contents

# A subproblem 2: matrix multiply

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

$$C \mathrel{\texttt{+=}} AB$$



# Formulate using subproblems

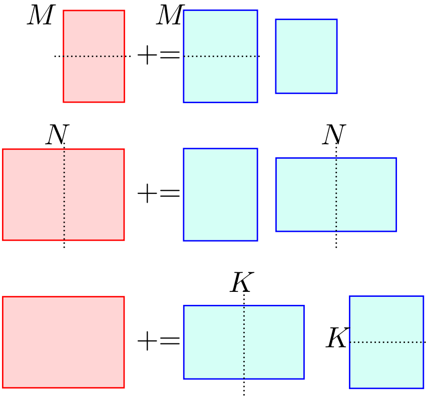
Three ways to decompose

- divide $M$ :
$$\begin{pmatrix} C_1 \\ C_2 \end{pmatrix} \texttt{ += } \begin{pmatrix} A_1 \\ A_2 \end{pmatrix} B$$
$\rightarrow C_1 \texttt{ += } A_1 B \;\; // \;\; C_2 \texttt{ += } A_2 B$
- divide $N$ :
$$\begin{pmatrix} C_1 & C_2 \end{pmatrix} \texttt{ += } A \begin{pmatrix} B_1 & B_2 \end{pmatrix}$$
$\rightarrow C_1 \texttt{ += } AB_1 \;\; // \;\; C_2 \texttt{ += } AB_2$
- divide $K$ :
$$C \texttt{ += } \begin{pmatrix} A_1 & A_2 \end{pmatrix} \begin{pmatrix} B_1 \\ B_2 \end{pmatrix}$$
$\rightarrow C \texttt{ += } A_1 B_1 \;\; ; \;\; C \texttt{ += } A_2 B_2$

# 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




```
gemm(A, B, C) {
  if ((M, N, K) = (1, 1, 1)) {
    c_11 += a_11 * b_11;
  } else if (M ≥ N and M ≥ K) {
    A_1, A_2 = split_h(A);
    C_1, C_2 = split_h(C);
    gemm(A_1, B, C_1);
    gemm(A_2, B, C_2);
  } else if (N ≥ M and N ≥ K)
    B_1, B_2 = split_v(B);
    C_1, C_2 = split_v(C);
    gemm(A, B_1, C_1);
    gemm(A, B_1, C_2);
  } else {
    A_1, A_2 = split_v(A);
    B_1, B_2 = split_h(B);
    gemm(A_1, B_1, C);
    gemm(A_2, B_2, C);
  }
}
```

# Where is parallelism in our example? Cholesky

1 /* *in place* */<br>
2 chol($A$) {<br>
3   **if** $(n = 1)$ $a_{11} := \sqrt{a_{11}}$;<br>
4   **else** {<br>
5     chol($A_{11}$);<br>
6     /* *triangular solve* */<br>
7     trsm($A_{11}, A_{12}$);<br>
8     $A_{21} = {}^{t}A_{12}$;<br>
9     $A_{22}$ -= $A_{21}A_{12}$<br>
10     chol($A_{22}$);<br>
11   }<br>
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!

```
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);
#pragma omp task
    gemm(A1, B, C1);
#pragma omp task
    gemm(A2, B, C2);
#pragma omp taskwait
  } else if (N ≥ M and N ≥ K)
    B1, B2 = split_v(B);
    C1, C2 = split_v(C);
#pragma omp task
    gemm(A, B1, C1);
#pragma omp task
    gemm(A, B1, C2);
#pragma omp taskwait
  } else {
    // same as before
    ...
  }
```

```
/* 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 {
#pragma omp task
    trsm(L, B1);
#pragma omp task
    trsm(L, B2);
#pragma omp taskwait
  }
}
```

# $T_1$ and $T_\infty$ of matrix multiply

```
gemm(A, B, C) {
  if ((M, N, K) = (1, 1, 1)) {
    c11 += a11 * b11;
  } else if (M ≥ N and M ≥ K) {
    ...
#pragma omp task
    gemm(A1, B, C1);
#pragma omp task
    gemm(A2, B, C2);
#pragma omp taskwait
  } else if (N ≥ M and N ≥ K)
    ...
#pragma omp task
    gemm(A, B1, C1);
#pragma omp task
    gemm(A, B1, C2);
#pragma omp taskwait
  } else {
    ...
    gemm(A1, B1, C);
    gemm(A2, B2, C);
  }
}
```

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

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

$$\Rightarrow \Theta(MNK)$$

# $T_1$ and $T_\infty$ of matrix multiply

```
gemm(A, B, C) {
  if ((M, N, K) = (1, 1, 1)) {
    c11 += a11 * b11;
  } else if (M ≥ N and M ≥ K) {
    ...
#pragma omp task
    gemm(A1, B, C1);
#pragma omp task
    gemm(A2, B, C2);
#pragma omp taskwait
  } else if (N ≥ M and N ≥ K)
    ...
#pragma omp task
    gemm(A, B1, C1);
#pragma omp task
    gemm(A, B1, C2);
#pragma omp taskwait
  } else {
    ...
    gemm(A1, B1, C);
    gemm(A2, B2, C);
  }
}
```

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

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

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

# $T_1$ and $T_\infty$ 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 {
#pragma omp task
    trsm(L, B1);
#pragma omp task
    trsm(L, B2);
#pragma omp taskwait
  }
}
```

Work ($T_1$), written by $W_{\mathrm{trsm}}(M, N) =$

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

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

# $T_1$ and $T_\infty$ of triangular solve

```
/* triangular solve LX = B.
   replace B with X */
trsm(L, B) {
  if (M = 1) {
    B /= l_11;
  } else if (M ≥ N) {
    trsm(L_11, B_1);
    B_2 -= L_21 B_1;
    trsm(L_22, B_2);
  } else {
#pragma omp task
    trsm(L, B_1);
#pragma omp task
    trsm(L, B_2);
#pragma omp taskwait
  }
}
```

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

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

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

# $T_1$ and $T_\infty$ of Cholesky

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

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

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

$\Rightarrow \Theta(n^3)$

# $T_1$ and $T_\infty$ of Cholesky

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

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

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

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

# 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 $T_\infty$
  - $T_1$ will decrease (good thing)
  - $T_\infty$ will increase due to a larger serial computation at leaves