

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

Kenjiro Taura

Contents

- 1 Introduction
- 2 An example : k -d tree construction
 - k -d tree
- 3 Parallelizing divide and conquer algorithms
- 4 Reasoning about speedup
 - Work and critical path length
 - Greedy scheduler theorem
 - Calculating work and critical path
- 5 More divide and conquer examples
 - Merge sort
 - Cholesky factorization
 - Triangular solve
 - Matrix multiply

Contents

- 1 Introduction
- 2 An example : k -d tree construction
 - k -d tree
- 3 Parallelizing divide and conquer algorithms
- 4 Reasoning about speedup
 - Work and critical path length
 - Greedy scheduler theorem
 - Calculating work and critical path
- 5 More divide and conquer examples
 - Merge sort
 - Cholesky factorization
 - Triangular solve
 - Matrix multiply

Goals

learn:

- the power of divide and conquer paradigm
- how to write and parallelize divide and conquer algorithms with task parallelism
- and how to reason about the speedup of task parallel programs
 - work
 - critical path length
 - Greedy Scheduler theorem

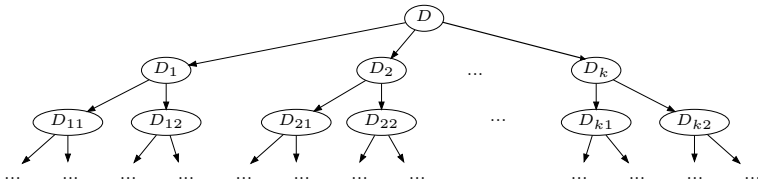
Divide and conquer algorithms

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

```

1  answer solve( $D$ ) {
2      if ( trivial( $D$ )) {
3          return trivially_solve( $D$ );
4      } else {
5           $D_1, \dots, D_k = \text{divide}(\mathbf{D})$ ; // divide the problem into sub problems
6           $a_1 = \text{solve}(D_1)$ ; ...;  $a_k = \text{solve}(D_k)$ ; // solve them
7          return combine( $a_1, \dots, a_k$ ); // 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 supports 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

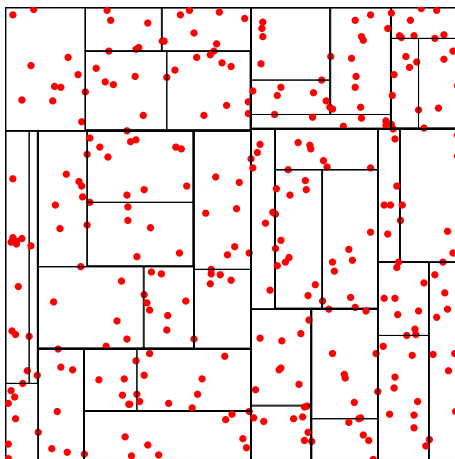
- 1 Introduction
- 2 An example : k -d tree construction
 - k -d tree
- 3 Parallelizing divide and conquer algorithms
- 4 Reasoning about speedup
 - Work and critical path length
 - Greedy scheduler theorem
 - Calculating work and critical path
- 5 More divide and conquer examples
 - Merge sort
 - Cholesky factorization
 - Triangular solve
 - Matrix multiply

Contents

- 1 Introduction
- 2 An example : k -d tree construction
 - k -d tree
- 3 Parallelizing divide and conquer algorithms
- 4 Reasoning about speedup
 - Work and critical path length
 - Greedy scheduler theorem
 - Calculating work and critical path
- 5 More divide and conquer examples
 - Merge sort
 - Cholesky factorization
 - Triangular solve
 - Matrix multiply

k -d tree

- A data structure to hierarchically organize points to facilitate “nearest neighbor” or “proximity” searches, usually in 2D or 3D space
- Each node represents a rectangular region



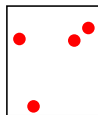
k -d tree construction

- 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

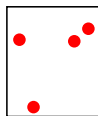
Leaf:



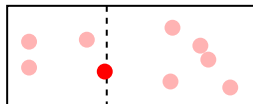
k -d tree construction

- 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:



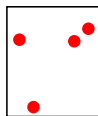
Internal:



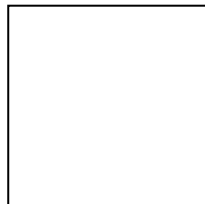
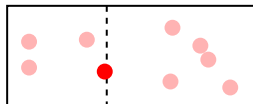
k -d tree construction

- 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:



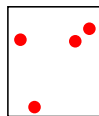
Internal:



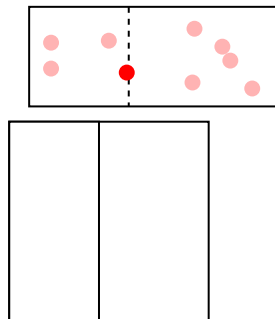
k -d tree construction

- 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)

Leaf:



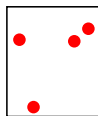
Internal:



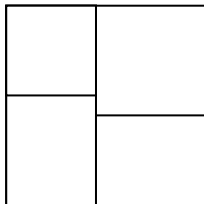
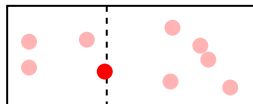
k -d tree construction

- 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:



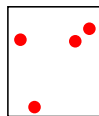
Internal:



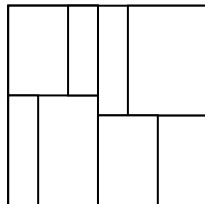
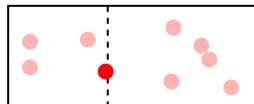
k -d tree construction

- 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)

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

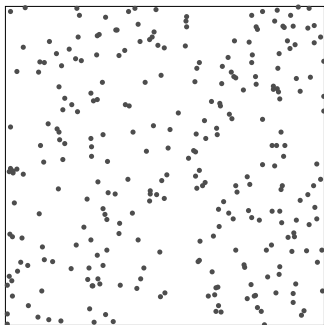
How to build a k -d tree

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

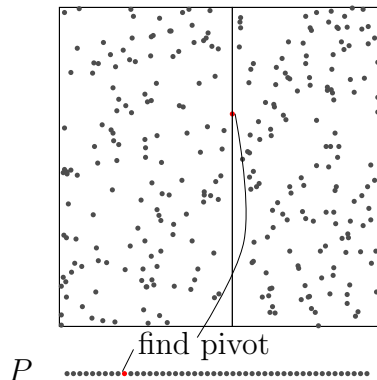
- to build a tree for a rectangle R and points P in R ,



P

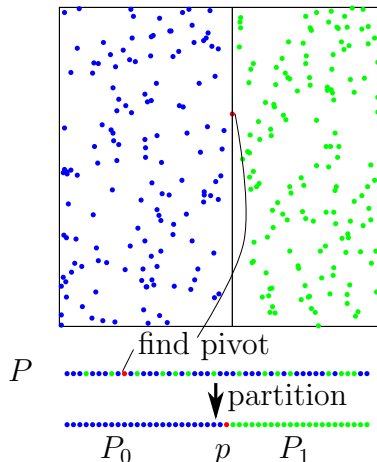
divide and conquer method

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



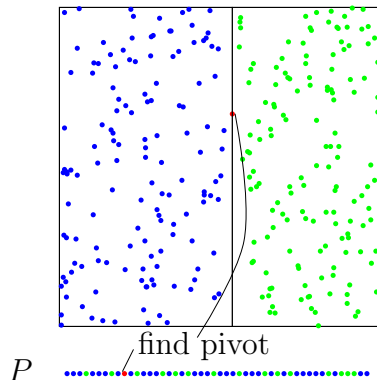
divide and conquer method

- 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 coordinate $< p$'s
 - P_1 : points whose x coordinate $\geq p$'s (except p)



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 coordinate $< p$'s
 - P_1 : points whose x coordinate $\geq p$'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         p = find_pivot(P, depth % D);
13         /* split R into two sub-rectangles */
14         R0,R1 = split_rect(R, depth % D, p.pos[depth % D]);
15         /* partition P by their coordinate lower/higher than p's coordinate */
16         P0,P1 = partition(P - { p }, depth % D, p.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

- $p = \text{find_pivot}(P, d)$
 - 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 $\Rightarrow O(1)$
- $R_0, R_1 = \text{split_rect}(R, d, c)$
 - split a rectangular region R by a (hyper-)plane “ d th coordinate = c ”
 - just make two rectangular regions $\Rightarrow O(1)$
- $P_0, P_1 = \text{partition}(P, d, c)$
 - 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

- 1 Introduction
- 2 An example : k -d tree construction
 - k -d tree
- 3 Parallelizing divide and conquer algorithms
- 4 Reasoning about speedup
 - Work and critical path length
 - Greedy scheduler theorem
 - Calculating work and critical path
- 5 More divide and conquer examples
 - Merge sort
 - Cholesky factorization
 - Triangular solve
 - Matrix multiply

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

```
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          p = find_pivot(P, depth % D);
8          R0,R1 = split_rect(R, depth % D, p.pos[depth % D]);
9          P0,P1 = partition(P - { p }, depth % D, p.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 }
```

- cumbersome to parallelize with only parallel loops

Note: tasks and GPU

- task parallelism (e.g., OpenMP `#pragma omp task`) is a great tool to implement divide-and-conquer, [on CPUs](#)
- [on GPUs](#), the implementation status is unclear and far from done
 - NVIDIA HPC SDK (nvc/nvc++) : not supported at all (writing `task` pragmas within `target` region results in compile-time error)
 - LLVM (clang/clang++) : compilation succeeds (at least for simple programs), but how tasks are distributed across CUDA-threads or SMs is uncertain
- there are inherent challenges around mapping dynamically created tasks onto the SIMT execution model of GPUs
- never expect it “just works” (more bluntly, avoid it altogether, at least for now)

Contents

- 1 Introduction
- 2 An example : k -d tree construction
 - k -d tree
- 3 Parallelizing divide and conquer algorithms
- 4 Reasoning about speedup
 - Work and critical path length
 - Greedy scheduler theorem
 - Calculating work and critical path
- 5 More divide and conquer examples
 - Merge sort
 - Cholesky factorization
 - Triangular solve
 - Matrix multiply

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

- 1 Introduction
- 2 An example : k -d tree construction
 - k -d tree
- 3 Parallelizing divide and conquer algorithms
- 4 Reasoning about speedup
 - Work and critical path length
 - Greedy scheduler theorem
 - Calculating work and critical path
- 5 More divide and conquer examples
 - Merge sort
 - Cholesky factorization
 - Triangular solve
 - Matrix multiply

Work and critical path length

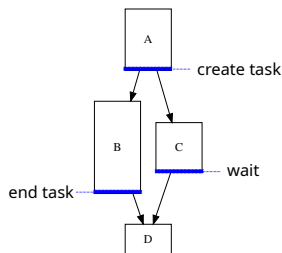
- **Work:** = the total amount of work of the computation
 - = the time it takes in a serial execution
- **Critical path length:** = the maximum length of dependent chain of computation
 - a more precise definition follows, with *computational DAGs*

Computational DAGs

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

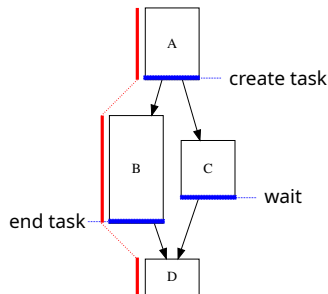
- a node = an interval of computation free of task parallel primitives
- an edge = a dependency between two nodes, of three types:
 - parent → created child
 - child → waiting parent
 - a node → the next node in the same task

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



A computational DAG and critical path length

- Consider each node is augmented with the time it takes 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



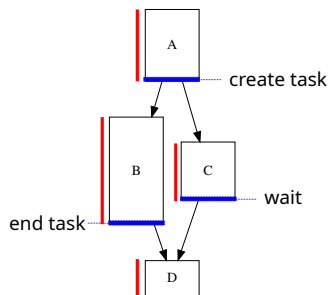
Given a computational DAG,

critical path length = the length of the longest path 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

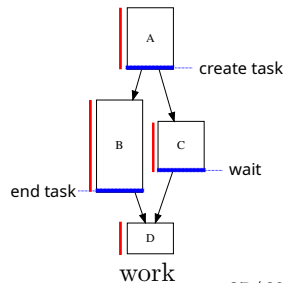
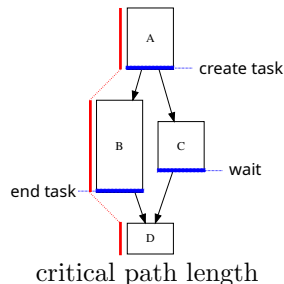
- Work, too, can be elegantly defined in light of computational DAGs



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



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

- 1 Introduction
- 2 An example : k -d tree construction
 - k -d tree
- 3 Parallelizing divide and conquer algorithms
- 4 Reasoning about speedup
 - Work and critical path length
 - Greedy scheduler theorem
 - Calculating work and critical path
- 5 More divide and conquer examples
 - Merge sort
 - Cholesky factorization
 - Triangular solve
 - Matrix multiply

The greedy scheduler theorem

- Assume:
 - you have P processors
 - they are *greedy*, in the sense that a processor is *always busy* on a node whenever there is *any runnable* node (a node whose predecessors have finished) 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 node whenever there is *any runnable* node (a node whose predecessors have finished) 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 node whenever there is *any runnable* node (a node whose predecessors have finished) 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!

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

A few facts that help you understand it

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,

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

- what a greedy scheduler achieves is

$$T_P \leq \min\left(\frac{T_1}{P}, T_\infty\right)$$

- 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_∞

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

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

or equivalently,

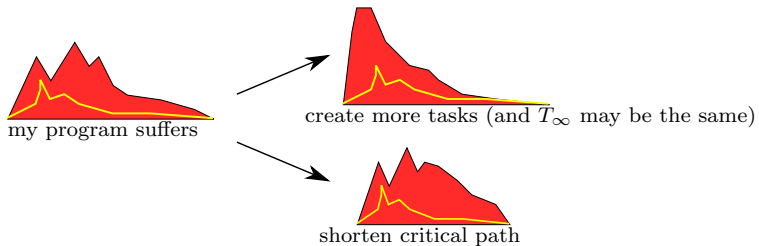
$$\frac{T_1}{T_\infty} \gg P$$

- 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

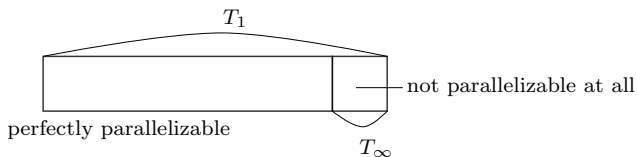
Suffer from low speedup? \Rightarrow try to shorten the critical path

*people are tempted to think creating **more and more tasks** is the way; they are useless, unless it shortens 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_∞), 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

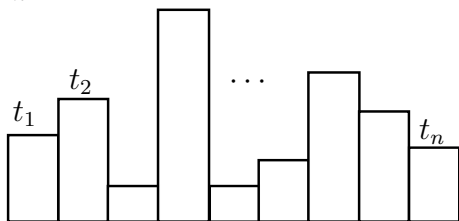
$$T_P \leq \frac{T_1 - T_\infty}{P} + T_\infty,$$

which coincides with what the greedy scheduler theorem says (for more general cases)

- takeaway: want to get a good speedup? \Rightarrow minimize T_∞ , or the work not parallelized

A special case (2) — “bag of tasks”

- assume we have a set of n independent tasks whose runtimes are t_1, t_2, \dots, t_n



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

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
- 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, as we will see below)

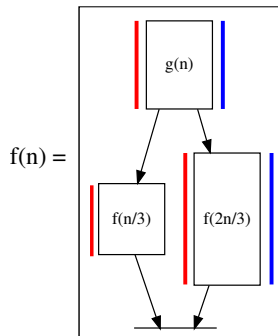
Contents

- 1 Introduction
- 2 An example : k -d tree construction
 - k -d tree
- 3 Parallelizing divide and conquer algorithms
- 4 Reasoning about speedup
 - Work and critical path length
 - Greedy scheduler theorem
 - Calculating work and critical path
- 5 More divide and conquer examples
 - Merge sort
 - Cholesky factorization
 - Triangular solve
 - Matrix multiply

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) $W_f(n) \leq W_g(n) + W_f(n/3) + W_f(2n/3)$
- (critical path) $C_f(n) \leq C_g(n) + \max\{C_f(n/3), C_f(2n/3)\}$
- 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     p = find_pivot(P, depth % D);
8     R0,R1 = split_rect(R, depth % D, p.pos[depth % D]);
9     P0,P1 = partition(P - { p }, depth % D, p.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  } }
```

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 $\alpha < 1$, 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     p = find_pivot(P, depth % D);  
8     R0,R1 = split_rect(R, depth % D, p.pos[depth % D]);  
9     P0,P1 = partition(P - { p }, depth % D, p.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  } }
```

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

omitting math,

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

Speedup of k -d tree construction

- Now we have:

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

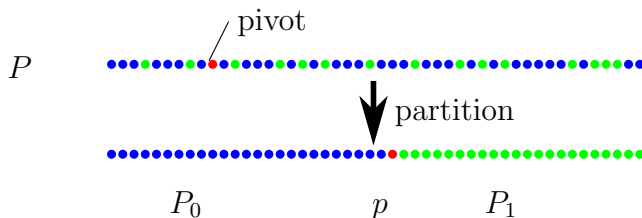
- \Rightarrow

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

- 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**



How to parallelize partition?

- it is a challenging exercise
- divide-and-conquer helps again!
- to partition an entire array $P[0 : n]$...

How to parallelize partition?

- it is a challenging exercise
- divide-and-conquer helps again!
- to partition an entire array $P[0 : n]$...
 - ① partition $P[0 : n/2]$ and $P[n/2 : n]$...

How to parallelize partition?

- it is a challenging exercise
- divide-and-conquer helps again!
- to partition an entire array $P[0 : n]$...
 - 1 partition $P[0 : n/2]$ and $P[n/2 : n]$...
 - 2 they need to be given

How to parallelize partition?

- it is a challenging exercise
- divide-and-conquer helps again!
- to partition an entire array $P[0 : n]$...
 - ① partition $P[0 : n/2]$ and $P[n/2 : n]$...
 - ② they need to be given
 - a separate destination array (say Q)

How to parallelize partition?

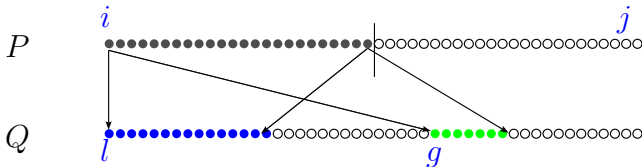
- it is a challenging exercise
- divide-and-conquer helps again!
- to partition an entire array $P[0 : n]$...
 - ① partition $P[0 : n/2]$ and $P[n/2 : n]$...
 - ② they need to be given
 - a separate destination array (say Q)
 - indexes to specify the part of P to partition

How to parallelize partition?

- it is a challenging exercise
- divide-and-conquer helps again!
- to partition an entire array $P[0 : n]$...
 - 1 partition $P[0 : n/2]$ and $P[n/2 : n]$...
 - 2 they need to be given
 - a separate destination array (say Q)
 - indexes to specify the part of P to partition
 - indexes to specify where to store left/right elements to

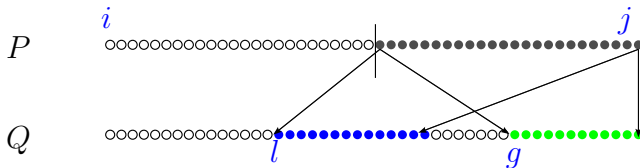
How to parallelize partition?

```
1  /* partition P[i:j], by putting
2     -      P0 -> Q[l:]; and
3     - {piv} + P1 -> Q[g:] */
4  void partition(P, Q, i, j, piv, l, g) {
5     if (j - i < threshold) { partition_seq(P, Q, i, j, piv, l, g); }
6     else {
7         long h = (i + j) / 2;
8         /* count the number of elements in P0 and P1 in the left half of P */
9         long dl, dg = count(P, i, h, piv);
10        create_task partition(A, B, i, h, piv, l,      g);
11                partition(A, B, h, j, piv, l + dl, g + dg);
12    }
13    wait;
14 }
```



How to parallelize partition?

```
1  /* partition P[i:j], by putting
2     -      P0 -> Q[l:]; and
3     - {piv} + P1 -> Q[g:] */
4  void partition(P, Q, i, j, piv, l, g) {
5     if (j - i < threshold) { partition_seq(P, Q, i, j, piv, l, g); }
6     else {
7         long h = (i + j) / 2;
8         /* count the number of elements in P0 and P1 in the left half of P */
9         long dl, dg = count(P, i, h, piv);
10        create_task partition(A, B, i, h, piv, l,      g);
11                partition(A, B, h, j, piv, l + dl, g + dg);
12    }
13    wait;
14 }
```



Critical Path of `partition`

let $C_{\text{partition}}(n)$ be the maximum critical path length for partitioning $\leq n$ elements

$$C_{\text{partition}}(n) \leq C_{\text{count}}(n/2) + C_{\text{partition}}(n/2)$$

- `count` can be easily parallelized (critical path $\in O(\log n)$)
- it is now easy to see

$$C_{\text{partition}}(n) \in \log^2 n$$

Contents

- 1 Introduction
- 2 An example : k -d tree construction
 - k -d tree
- 3 Parallelizing divide and conquer algorithms
- 4 Reasoning about speedup
 - Work and critical path length
 - Greedy scheduler theorem
 - Calculating work and critical path
- 5 More divide and conquer examples
 - Merge sort
 - Cholesky factorization
 - Triangular solve
 - Matrix multiply

Contents

- 1 Introduction
- 2 An example : k -d tree construction
 - k -d tree
- 3 Parallelizing divide and conquer algorithms
- 4 Reasoning about speedup
 - Work and critical path length
 - Greedy scheduler theorem
 - Calculating work and critical path
- 5 More divide and conquer examples
 - Merge sort
 - Cholesky factorization
 - Triangular solve
 - Matrix multiply

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

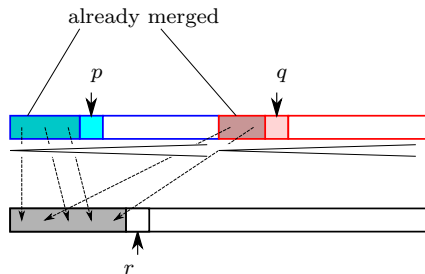
$$\begin{aligned}T_1 &= W_{\text{ms}}(n) \in \Theta(n \log n), \\T_\infty &= C_{\text{ms}}(n) \in \Theta(n).\end{aligned}$$

the average parallelism

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

How (serial) merge works

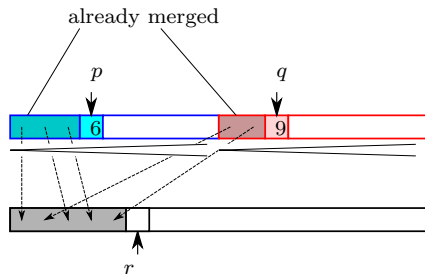
```
/* merge a_beg ... a_end
   and b_beg ... b_end
   into c */
void
merge(elem * a, elem * a_end,
      elem * b, elem * b_end,
      elem * c) {
    elem * p = a, * q = b, * r = c;
    while (p < a_end && q < b_end) {
        if (*p < *q) { *r++ = *p++; }
        else { *r++ = *q++; }
    }
    while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



Looks very serial ...

How (serial) merge works

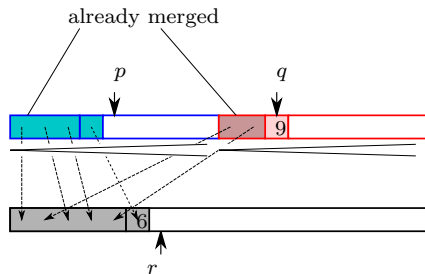
```
/* merge a_beg ... a_end
   and b_beg ... b_end
   into c */
void
merge(elem * a, elem * a_end,
      elem * b, elem * b_end,
      elem * c) {
    elem * p = a, * q = b, * r = c;
    while (p < a_end && q < b_end) {
        if (*p < *q) { *r++ = *p++; }
        else { *r++ = *q++; }
    }
    while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

- 1 Introduction
- 2 An example : k -d tree construction
 - k -d tree
- 3 Parallelizing divide and conquer algorithms
- 4 Reasoning about speedup
 - Work and critical path length
 - Greedy scheduler theorem
 - Calculating work and critical path
- 5 More divide and conquer examples
 - Merge sort
 - Cholesky factorization
 - Triangular solve
 - Matrix multiply

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)

The diagram illustrates the Cholesky factorization equation $A = L {}^tL$. On the left, a square matrix A is shown with a blue border. The width is labeled n above and the height is labeled n to the left. In the center of the square is the letter A . This is followed by an equals sign. To the right of the equals sign are two square matrices. The first is a lower triangular matrix L with a red border and an orange-shaded lower triangle; the letter L is in the center. The second is its transpose tL , also with a red border and an orange-shaded upper triangle; the label tL is in the center of the shaded area.

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

- ❶ $A_{11} = L_{11} {}^tL_{11}$
- ❷ ${}^tA_{21} = L_{11} {}^tL_{21}$
- ❸ $A_{22} = L_{21} {}^tL_{21} + L_{22} {}^tL_{22}$

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 ${}^tA_{21} = \textcolor{red}{L}_{11} {}^tL_{21}$

③ $A_{22} = \textcolor{red}{L}_{21} {}^t\textcolor{red}{L}_{21} + L_{22} {}^tL_{22}$

```
1  /* Cholesky factorization */
2  chol(A) {
3      if (n == 1) return (sqrt(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
10                   L21  L22 )
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 (sqrt(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
10                   L21  L22 )
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 (sqrt(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
10                   L21  L22 )
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 (sqrt(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
10                  ( L21  L22 )
11                )
12      }
```

Remark 1 : “In-place update” version

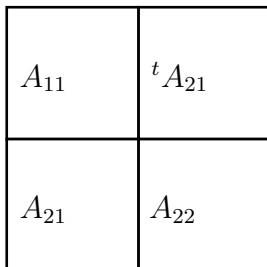
- Instead of returning the answer as another matrix, it is often possible to update the input matrix with the answer
- When possible, it is desirable, as it avoids extra copies

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

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

In-place Cholesky at work

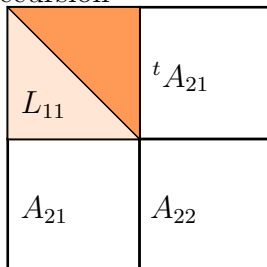
```
/* in place */  
chol(A) {  
  if (n == 1) a11 :=  $\sqrt{a_{11}}$ ;  
  else {  
    chol(A11);  
    /* triangular solve */  
    trsm(A11, A12);  
    A21 =  ${}^t A_{12}$ ;  
    A22 -= A21 A12  
    chol(A22);  
  }  
}
```



In-place Cholesky at work

```
/* in place */  
chol(A) {  
  if (n == 1) a11 :=  $\sqrt{a_{11}}$ ;  
  else {  
    chol(A11);  
    /* triangular solve */  
    trsm(A11, A12);  
    A21 =  ${}^t A_{12}$ ;  
    A22 -= A21A12  
    chol(A22);  
  }  
}
```

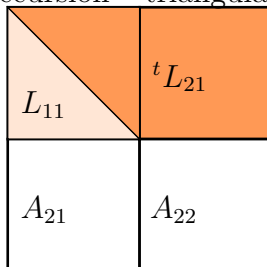
recursion



In-place Cholesky at work

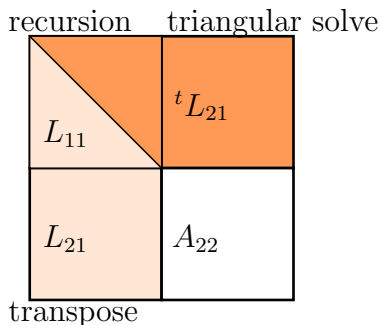
```
/* in place */  
chol(A) {  
  if (n == 1)  $a_{11} := \sqrt{a_{11}}$ ;  
  else {  
    chol( $A_{11}$ );  
    /* triangular solve */  
    trsm( $A_{11}$ ,  $A_{12}$ );  
     $A_{21} = {}^t A_{12}$ ;  
     $A_{22} -= A_{21} A_{12}$   
    chol( $A_{22}$ );  
  }  
}
```

recursion triangular solve



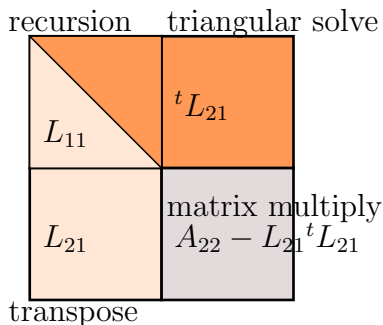
In-place Cholesky at work

```
/* 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( $A_{22}$ );  
  }  
}
```



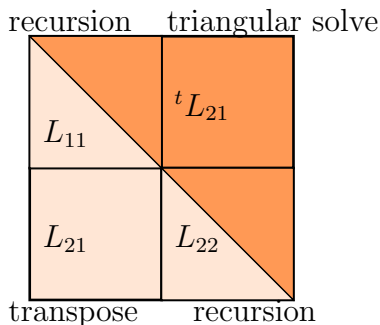
In-place Cholesky at work

```
/* in place */  
chol(A) {  
  if (n == 1) a11 :=  $\sqrt{a_{11}}$ ;  
  else {  
    chol(A11);  
    /* triangular solve */  
    trsm(A11, A12);  
    A21 = tA12;  
    A22 -= A21A12  
    chol(A22);  
  }  
}
```



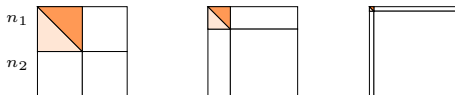
In-place Cholesky at work

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



Remark 2 : where to decompose

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

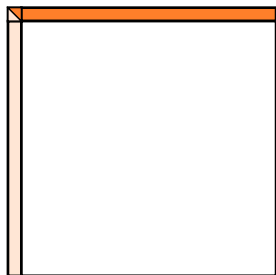


```
/* general */
chol(A) {
  if (n = 1)  $a_{11} := \sqrt{a_{11}}$ ;
  else {
    chol( $A_{11}$ );
    /* triangular solve */
    trsm( $A_{11}$ ,  $A_{12}$ );
     $A_{21} = {}^t A_{12}$ ;
     $A_{22} -= A_{21} A_{12}$ ;
    chol( $A_{22}$ );
  }
}
```

```
1 /* loop-like */
2 chol(A) {
3   if (n = 1)  $a_{11} := \sqrt{a_{11}}$ ;
4   else {
5      $a_{11} := \sqrt{a_{11}}$ ;
6     /* triangular solve */
7      $\vec{a}_{12} /= a_{11}$ ;
8      $\vec{a}_{21} /= a_{11}$ ;
9      $A_{22} -= \vec{a}_{21} \vec{a}_{12}$ ;
10    chol( $A_{22}$ );
11  }
12 }
```

Recursion to loops

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

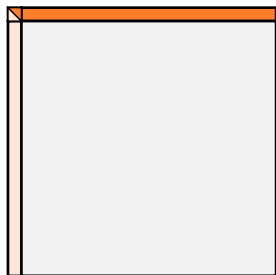


```
1  /* loop version */
2  chol_loop(A) {
3      for (k = 1; k ≤ n; k ++ ) {
4          akk := √akk;
5          Ak,k+1:n /= akk;
6          Ak+1:n,k /= akk;
7          Ak+1:n,k+1:n -= Ak:n,kAk,k:n
8      }
9  }
```

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

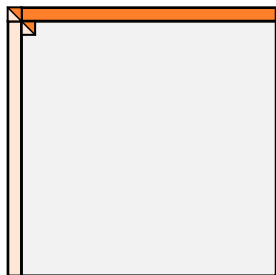


```
1  /* loop version */
2  chol_loop(A) {
3      for (k = 1; k ≤ n; k ++ ) {
4          akk := √akk;
5          Ak,k+1:n /= akk;
6          Ak+1:n,k /= akk;
7          Ak+1:n,k+1:n -= Ak:n,kAk,k:n
8      }
9  }
```

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

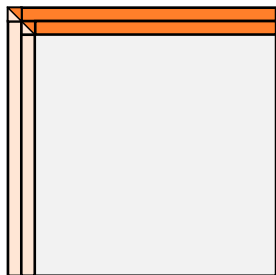


```
1  /* loop version */
2  chol_loop(A) {
3      for (k = 1; k ≤ n; k ++ ) {
4           $a_{kk} := \sqrt{a_{kk}}$ ;
5           $A_{k, k+1:n} /= a_{kk}$ ;
6           $A_{k+1:n, k} /= a_{kk}$ ;
7           $A_{k+1:n, k+1:n} -= A_{k:n, k} A_{k, k:n}$ 
8      }
9  }
```

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

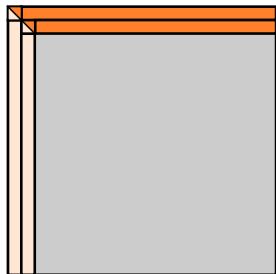


```
1  /* loop version */
2  chol_loop(A) {
3      for (k = 1; k ≤ n; k ++ ) {
4          a_kk := √a_kk;
5          A_k,k+1:n /= a_kk;
6          A_k+1:n,k /= a_kk;
7          A_k+1:n,k+1:n -= A_k:n,k A_k,k:n
8      }
9  }
```

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

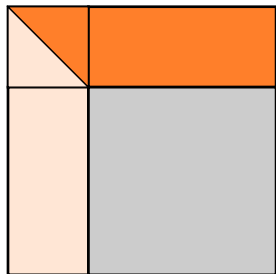


```
1  /* loop version */
2  chol_loop(A) {
3      for (k = 1; k ≤ n; k ++ ) {
4          akk := √akk;
5          Ak,k+1:n /= akk;
6          Ak+1:n,k /= akk;
7          Ak+1:n,k+1:n -= Ak:n,kAk,k:n
8      }
9  }
```

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

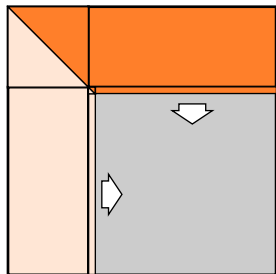


```
1  /* loop version */
2  chol_loop(A) {
3      for (k = 1; k ≤ n; k ++ ) {
4          akk := √akk;
5          Ak,k+1:n /= akk;
6          Ak+1:n,k /= akk;
7          Ak+1:n,k+1:n -= Ak:n,kAk,k:n
8      }
9  }
```

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          akk := √akk;  
5          Ak,k+1:n /= akk;  
6          Ak+1:n,k /= akk;  
7          Ak+1:n,k+1:n -= Ak:n,kAk,k:n  
8      }  
9  }
```

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

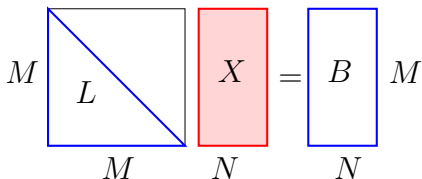
Contents

- 1 Introduction
- 2 An example : k -d tree construction
 - k -d tree
- 3 Parallelizing divide and conquer algorithms
- 4 Reasoning about speedup
 - Work and critical path length
 - Greedy scheduler theorem
 - Calculating work and critical path
- 5 More divide and conquer examples
 - Merge sort
 - Cholesky factorization
 - **Triangular solve**
 - Matrix multiply

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

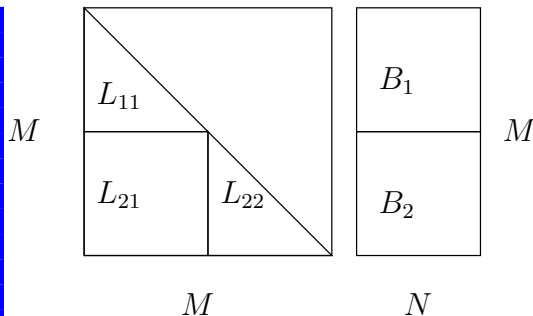
$$L \begin{pmatrix} X_1 & X_2 \end{pmatrix} = \begin{pmatrix} B_1 & B_2 \end{pmatrix} \Rightarrow$$

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 -= L21B1;
9          trsm(L22, B2);
10     } else {
11         trsm(L, B1);
12         trsm(L, B2);
13     }
14 }
```

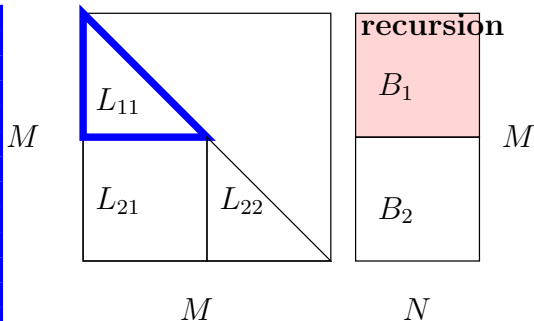
Triangular solve at work

```
/* triangular solve  $LX = B$ .  
   replace  $B$  with  $X$  */  
trsm( $L, B$ ) {  
  if ( $M = 1$ ) {  
     $B \neq l_{11}$ ;  
  } else if ( $M \geq N$ ) {  
    trsm( $L_{11}, B_1$ );  
     $B_2 \mathrel{:=} L_{21}B_1$ ;  
    trsm( $L_{22}, B_2$ );  
  } else {  
    trsm( $L, B_1$ );  
    trsm( $L, B_2$ );  
  }  
}
```



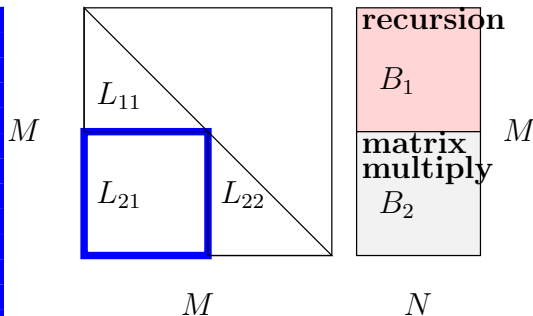
Triangular solve at work

```
/* triangular solve  $LX = B$ .  
   replace  $B$  with  $X$  */  
trsm( $L, B$ ) {  
  if ( $M = 1$ ) {  
     $B \neq l_{11}$ ;  
  } else if ( $M \geq N$ ) {  
    trsm( $L_{11}, B_1$ );  
     $B_2 \mathrel{:=} L_{21}B_1$ ;  
    trsm( $L_{22}, B_2$ );  
  } else {  
    trsm( $L, B_1$ );  
    trsm( $L, B_2$ );  
  }  
}
```



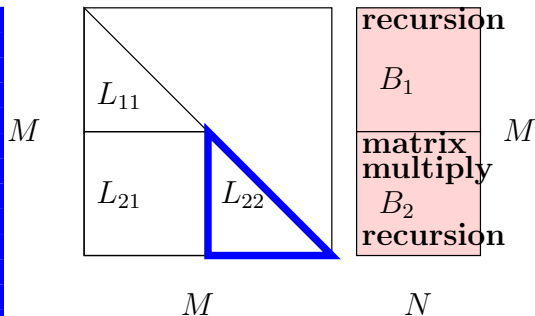
Triangular solve at work

```
/* triangular solve  $LX = B$ .  
   replace  $B$  with  $X$  */  
trsm( $L, B$ ) {  
  if ( $M = 1$ ) {  
     $B \neq l_{11}$ ;  
  } else if ( $M \geq N$ ) {  
    trsm( $L_{11}, B_1$ );  
     $B_2 \mathrel{:=} L_{21}B_1$ ;  
    trsm( $L_{22}, B_2$ );  
  } else {  
    trsm( $L, B_1$ );  
    trsm( $L, B_2$ );  
  }  
}
```



Triangular solve at work

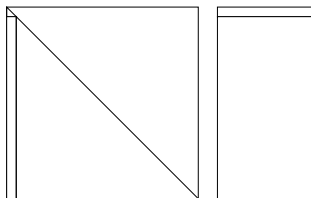
```
/* triangular solve  $LX = B$ .  
   replace  $B$  with  $X$  */  
trsm( $L, B$ ) {  
  if ( $M = 1$ ) {  
     $B \neq l_{11}$ ;  
  } else if ( $M \geq N$ ) {  
    trsm( $L_{11}, B_1$ );  
     $B_2 \mathrel{:=} L_{21}B_1$ ;  
    trsm( $L_{22}, B_2$ );  
  } else {  
    trsm( $L, B_1$ );  
    trsm( $L, B_2$ );  
  }  
}
```



Recursions and loops

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

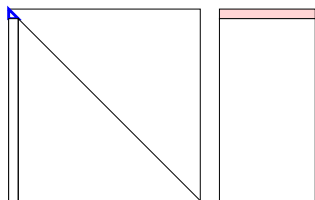
```
/* loop */  
trsm(L, B) {  
  for (k = 1; k ≤ M; k ++ ) {  
     $B_{k,1:M} \leftarrow l_{kk};$   
     $B_{k+1:M,1:M} -= L_{k+1:M,k} B_{k,1:M};$   
  }  
}
```



Recursions and loops

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

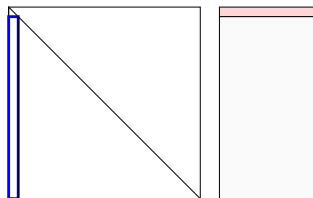
```
/* loop */  
trsm(L, B) {  
  for (k = 1; k ≤ M; k ++ ) {  
     $B_{k,1:M} \leftarrow l_{kk};$   
     $B_{k+1:M,1:M} \leftarrow L_{k+1:M,k} B_{k,1:M};$   
  }  
}
```



Recursions and loops

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

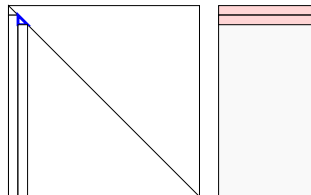
```
/* loop */  
trsm(L, B) {  
  for (k = 1; k ≤ M; k ++ ) {  
    Bk,1:M /= lkk;  
    Bk+1:M,1:M -= Lk+1:M,kBk,1:M;  
  }  
}
```



Recursions and loops

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

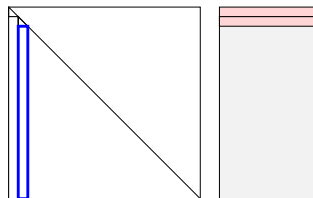
```
/* loop */  
trsm(L, B) {  
  for (k = 1; k ≤ M; k ++ ) {  
     $B_{k,1:M} \neq l_{kk}$ ;  
     $B_{k+1:M,1:M} -= L_{k+1:M,k} B_{k,1:M}$ ;  
  }  
}
```



Recursions and loops

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

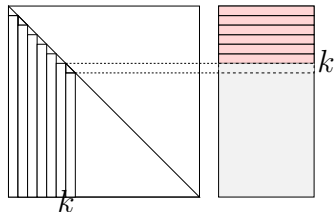
```
/* loop */  
trsm(L, B) {  
  for (k = 1; k ≤ M; k ++ ) {  
    Bk,1:M /= lkk;  
    Bk+1:M,1:M -= Lk+1:M,kBk,1:M;  
  }  
}
```



Recursions and loops

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

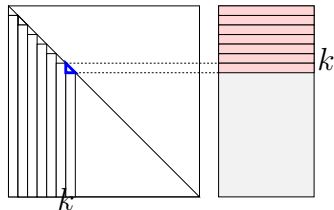
```
/* loop */  
trsm(L, B) {  
  for (k = 1; k ≤ M; k ++ ) {  
    Bk,1:M /= lkk;  
    Bk+1:M,1:M -= Lk+1:M,kBk,1:M;  
  }  
}
```



Recursions and loops

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

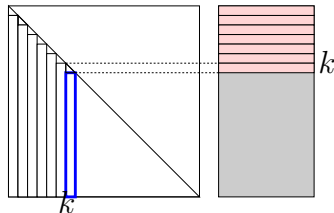
```
/* loop */  
trsm(L, B) {  
  for (k = 1; k ≤ M; k ++ ) {  
    Bk,1:M /= lkk;  
    Bk+1:M,1:M -= Lk+1:M,kBk,1:M;  
  }  
}
```



Recursions and loops

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

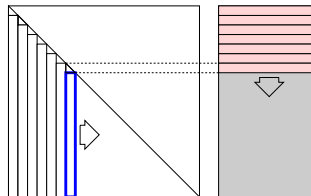
```
/* loop */  
trsm(L, B) {  
  for (k = 1; k ≤ M; k ++ ) {  
     $B_{k,1:M} \leftarrow l_{kk};$   
     $B_{k+1:M,1:M} \leftarrow L_{k+1:M,k} B_{k,1:M};$   
  }  
}
```



Recursions and loops

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

```
/* loop */  
trsm(L, B) {  
  for (k = 1; k ≤ M; k ++ ) {  
    Bk,1:M /= lkk;  
    Bk+1:M,1:M -= Lk+1:M,kBk,1:M;  
  }  
}
```

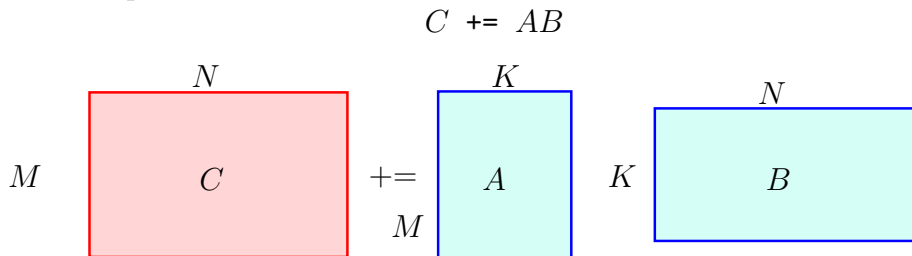


Contents

- 1 Introduction
- 2 An example : k -d tree construction
 - k -d tree
- 3 Parallelizing divide and conquer algorithms
- 4 Reasoning about speedup
 - Work and critical path length
 - Greedy scheduler theorem
 - Calculating work and critical path
- 5 More divide and conquer examples
 - Merge sort
 - Cholesky factorization
 - Triangular solve
 - Matrix multiply

A subproblem 2: matrix multiply

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



Formulate using subproblems

Three ways to decompose

- divide M :

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

$$\rightarrow C_1 += A_1 B \quad // \quad C_2 += A_2 B$$

- divide N :

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

$$\rightarrow C_1 += AB_1 \quad // \quad C_2 += AB_2$$

- divide K :

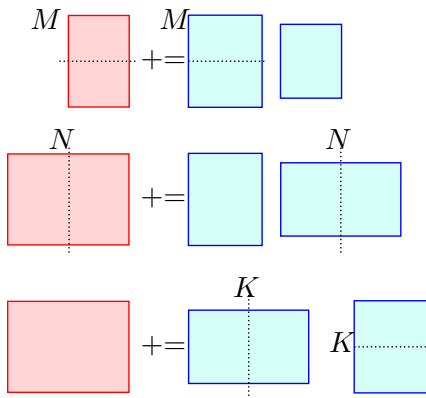
$$C += \begin{pmatrix} A_1 & A_2 \end{pmatrix} \begin{pmatrix} B_1 \\ B_2 \end{pmatrix}$$

$$\rightarrow C += A_1 B_1 \quad ; \quad C += A_2 B_2$$

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       $c_{11} += a_{11} * b_{11};$   
4    } else if (M ≥ N and M ≥ K) {  
5       $A_1, A_2 = \text{split\_h}(A);$   
6       $C_1, C_2 = \text{split\_h}(C);$   
7      gemm( $A_1, B, C_1$ );  
8      gemm( $A_2, B, C_2$ );  
9    } else if (N ≥ M and N ≥ K)  
10      $B_1, B_2 = \text{split\_v}(B);$   
11      $C_1, C_2 = \text{split\_v}(C);$   
12     gemm( $A, B_1, C_1$ );  
13     gemm( $A, B_1, C_2$ );  
14   } else {  
15      $A_1, A_2 = \text{split\_v}(A);$   
16      $B_1, B_2 = \text{split\_h}(B);$   
17     gemm( $A_1, B_1, C$ );  
18     gemm( $A_2, B_2, C$ );  
19   }  
20 }
```


Where is parallelism in our example?

Cholesky

```
1  /* in place */
2  chol(A) {
3      if (n = 1)  $a_{11} := \sqrt{a_{11}}$ ;
4      else {
5          chol( $A_{11}$ );
6          /* triangular solve */
7          trsm( $A_{11}$ ,  $A_{12}$ );
8           $A_{21} = {}^t A_{12}$ ;
9           $A_{22} -= A_{21} A_{12}$ 
10         chol( $A_{22}$ );
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       $B /= l_{11}$ ;  
6    } else if ( $M \geq N$ ) {  
7      trsm( $L_{11}$ ,  $B_1$ );  
8       $B_2 -= L_{21}B_1$ ;  
9      trsm( $L_{22}$ ,  $B_2$ );  
10   } else {  
11     trsm( $L$ ,  $B_1$ );  
12     trsm( $L$ ,  $B_2$ );  
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, B2, 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          A1, A2 = split_h(A);
6          C1, C2 = split_h(C);
7      #pragma omp task
8          gemm(A1, B, C1);
9      #pragma omp task
10         gemm(A2, B, C2);
11     #pragma omp taskwait
12     } else if (N ≥ M and N ≥ K)
13         B1, B2 = split_v(B);
14         C1, C2 = split_v(C);
15     #pragma omp task
16         gemm(A, B1, C1);
17     #pragma omp task
18         gemm(A, B2, C2);
19     #pragma omp taskwait
20     } else {
21         // same as before
22         ...
23     }
```

```
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 -= L21B1;
9          trsm(L22, B2);
10     } else {
11     #pragma omp task
12         trsm(L, B1);
13     #pragma omp task
14         trsm(L, B2);
15     #pragma omp taskwait
16     }
17 }
```

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) {  
        ...  
#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

$W_{\text{gemm}}(M, N, K) =$

$$\left\{ \begin{array}{l} \Theta(1) \\ ((M, N, K) = (1, 1, 1)) \\ 2W_{\text{gemm}}(M/2, N, K) + \Theta(1) \\ (M \text{ is largest}) \\ 2W_{\text{gemm}}(M, N/2, K) + \Theta(1) \\ (N \text{ is largest}) \\ 2W_{\text{gemm}}(M, N, K/2) + \Theta(1) \\ (K \text{ is largest}) \end{array} \right.$$

$$\Rightarrow \Theta(MNK)$$

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) {  
    ...  
#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_∞), written by

$C_{\text{gemm}}(M, N, K) =$

$$\left\{ \begin{array}{l} \Theta(1) \\ ((M, N, K) = (1, 1, 1)), \\ C_{\text{gemm}}(M/2, N, K) + \Theta(1) \\ (M \text{ is largest}) \\ C_{\text{gemm}}(M, N/2, K) + \Theta(1) \\ (N \text{ is largest}) \\ 2C_{\text{gemm}}(M, N, K/2) + \Theta(1) \\ (K \text{ is largest}) \end{array} \right.$$

$\Rightarrow \Theta(\log M + \log N + K)$ (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 ( $M = 1$ ) {  
         $B /= l_{11}$ ;  
    } else if ( $M \geq N$ ) {  
        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  
    }  
}
```

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 ( $M = 1$ ) {  
         $B /= l_{11}$ ;  
    } else if ( $M \geq N$ ) {  
        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_∞), written by
 $C_{\text{trsm}}(M, N) =$

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

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

T_1 and T_∞ of Cholesky

```
chol(A) {  
  if (n = 1) a11 :=  $\sqrt{a_{11}}$ ;  
  else {  
    chol(A11);  
    /* triangular solve */  
    trsm(A11, A12);  
    A21 = tA12;  
    A22 -= A21A12  
    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_∞ of Cholesky

```
chol(A) {  
  if (n = 1) a11 :=  $\sqrt{a_{11}}$ ;  
  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