Matrix algorithms: fast, stable, communication-optimizing—random?!

Ioana Dumitriu

Department of Mathematics
University of Washington (Seattle)

Joint work with Grey Ballard, James Demmel, Olga Holtz, Robert Kleinberg

IPAM: Convex Optimization and Algebraic Geometry
September 28, 2010

1. Motivation and Background
   - Main Concerns in Numerical Linear Algebra
   - Size issues
2. Reducing the flops
   - Stability of FMM
   - Stability of FLA
   - Why Randomize?
3. Reducing communication
   - Why is communication bad?
   - Randomized Spectral Divide and Conquer
   - Communication Costs
4. Conclusions

The big four issues in matrix algorithms.

- Accuracy of computation (how "far" the computed values are from the actual ones, in the presence of rounding error).

The big four issues in matrix algorithms.

- Accuracy of computation (how "far" the computed values are from the actual ones, in the presence of rounding error).
- Stability (how much the computed result will change if we perturb the problem a little bit).

The big four issues in matrix algorithms.

- Accuracy of computation (how "far" the computed values are from the actual ones, in the presence of rounding error).
- Stability (how much the computed result will change if we perturb the problem a little bit).
- Speed/flops complexity (how many operations they require, e.g., multiplications).

The big four issues in matrix algorithms.

- Accuracy of computation (how "far" the computed values are from the actual ones, in the presence of rounding error).
- Stability (how much the computed result will change if we perturb the problem a little bit).
- Speed/flops complexity (how many operations they require, e.g., multiplications).
- Parallelism (how to distribute the computation to multiple processors in order to optimize speed – if possible).

The big five issues in matrix algorithms.

- Accuracy of computation (how "far" the computed values are from the actual ones, in the presence of rounding error).
- Stability (how much the computed result will change if we perturb the problem a little bit).
- Speed/flops complexity (how many operations they require, e.g., multiplications).
- Parallelism (how to distribute the computation to multiple processors in order to optimize speed – if possible).
- Communication complexity (how to minimize the amount of back-and-forth between levels of memory or processors).

Size matters.

- We're concerned with industrial-sized problems and asymptotic complexities.
- Constants in $O(\cdot)$ terms may be large enough to make "everyday" use impractical.
- Some problems will reduce down to "manageable" size... then apply classical algorithms.

Fast matrix multiplication

- What's the minimum number of operations needed to multiply two matrices?
  - LinAlg 101 student: $O(n^3)$

Fast matrix multiplication

- What's the minimum number of scalar multiplications needed to multiply two matrices?
  - LinAlg 101 student: $O(n^3)$

Fast matrix multiplication

- What's the minimum number of scalar multiplications needed to multiply two matrices?
  - LinAlg 101 student: $O(n^3)$
  - ThCS/LinAlg 501 student: far fewer...

Fast matrix multiplication

- What's the minimum number of scalar multiplications needed to multiply two matrices?
  - LinAlg 101 student: $O(n^3)$
  - ThCS/LinAlg 501 student: far fewer...
- Late '60s (Strassen): at most $c \cdot n^{\log_2 7}$ are needed

Fast matrix multiplication

- What's the minimum number of scalar multiplications needed to multiply two matrices?
  - LinAlg 101 student: $O(n^3)$
  - ThCS/LinAlg 501 student: far fewer...

- Late '60s (Strassen): at most $c \cdot n^{\log_2 7}$ are needed
- Early 90's (Coppersmith-Winograd) at most $c \cdot n^{2.376}$ are needed

Fast matrix multiplication

- What's the minimum number of scalar multiplications needed to multiply two matrices?
  - LinAlg 101 student: $O(n^3)$
  - ThCS/LinAlg 501 student: far fewer...

- Late '60s (Strassen): at most $c \cdot n^{\log_2 7}$ are needed
- Early 90's (Coppersmith-Winograd) at most $c \cdot n^{2.376}$ are needed
- ... but is it true that only $c \cdot n^{2+\epsilon}$ are needed?...

Fast matrix multiplication

- What's the minimum number of scalar multiplications needed to multiply two matrices?
  - LinAlg 101 student: $O(n^3)$
  - ThCS/LinAlg 501 student: far fewer...

- Late '60s (Strassen): at most $c \cdot n^{\log_2 7}$ are needed
- Early 90's (Coppersmith-Winograd) at most $c \cdot n^{2.376}$ are needed
- ... but is it true that only $c \cdot n^{2+\epsilon}$ are needed?...

Matrix multiplication exponent: ω is the smallest number such that there exist fast matrix multiplication algorithms running in $O(n^{\omega+\epsilon})$ time for any $\epsilon > 0$; $2.376 \geq \omega \geq 2$.

Fast matrix multiplication is stable!

- No concrete progress in the '00s, but new promising paths via group-theoretic algorithms (Cohn, Kleinberg, Szegedy, & Umans)
- We may not know what ω is, but we *do* now know that the algorithm that achieves it does so *stably*. (Demmel, D., Holtz, & Kleinberg '07)
- In other words, errors in computation (due to roundoff) will not accumulate to the point where the calculation is completely unreliable, and small changes in the problem will not incur big changes in the result.
- No algorithm running in less than $O(n^3)$ can be entry-wise stable (Miller '75), but we achieve norm-stability.

Fast matrix multiplication is stable!

- Proved (Demmel, D., Holtz, Kleinberg '07) that
  - Cohn, Umans et al algorithms are norm-stable.
  - Bilinear non-commutative algorithms are norm-stable.
  - Combined this with result that the exponent of matrix multiplication is achieved by bilinear non-commutative algorithms (folklore; Raz '03).
  - Therefore *the exponent of matrix multiplication can be achieved by stable algorithms.*

- What about other linear algebra operations?

Fast linear algebra is stable?

- Assume now a **"black box"** algorithm for fast and stable matrix multiplication (any exponent $3 > \gamma \geq \omega$). Can we use it to do other **linear algebra** operations fast and stably?
- Known: complexity of (virtually all) Linear Algebra = complexity of MM.

Fast linear algebra is stable?

- Assume now a **"black box"** algorithm for fast and stable matrix multiplication (any exponent $3 > \gamma \geq \omega$). Can we use it to do other **linear algebra** operations fast and stably?
- Known: complexity of (virtually all) Linear Algebra = complexity of MM.
- *Not known* (previously): how to achieve it *stably*.
- Must use blocked algorithms... *new* block algorithms (the old can do the trick).

Fast linear algebra is stable!

- Proved (Demmel, D., Holtz, '07) that FLA is stable: e.g., LU, QR, EIG/SVD, inversion, Sylvester eqs, $A\backslash b$, etc.

- Essentially, **MM**, **LU**, and **QR** will get you everything else.
- For some ops (e.g. LU, matrix inversion) we need to raise bit complexity to achieve backward stability (need to count *bit operations*); others ops (QR, EIG/SVD) work in arithmetic operations.

Fast (randomized) linear algebra is stable!

- Proved (Demmel, D., Holtz, '07) that FLA is stable: e.g., LU, QR, EIG/SVD, inversion, Sylvester eqs, $A\backslash b$, etc.

- Most of these are deterministic, but we need to randomize to be able to do EIG/SVD.
- In particular, we need a *randomized* rank-revealing factorization to do Divide and Conquer (both for symmetric and non-symmetric case).
- All are parallelizable!

How do we find the rank of a matrix–fast?

- SVD.

How do we find the rank of a matrix–fast?

- SVD. Can't use, circular.

How do we find the rank of a matrix–fast?

- SVD. Can't use, circular.
- QR, no pivoting.

How do we find the rank of a matrix–fast?

- SVD. Can't use, circular.
- QR, no pivoting. If matrix singular, will break.

How do we find the rank of a matrix–fast?

- SVD. Can't use, circular.
- QR, no pivoting. If matrix singular, will break.
- QR with pivoting.

How do we find the rank of a matrix–fast?

- SVD. Can't use, circular.
- QR, no pivoting. If matrix singular, will break.
- QR with pivoting. Worst-case $O(n^4)$!

How do we find the rank of a matrix–fast?

- SVD. Can't use, circular.
- QR, no pivoting. If matrix singular, will break.
- QR with pivoting. Worst-case $O(n^4)$!
- Use a random orthogonal (Haar) matrix to "muddle" columns!

Randomized URV (**RURV**)

Suppose want to find $rank(A)$ in exact arithmetic.

- Construct Haar (random orthogonal/unitary) matrix V;
- $[U, R] = qr(A \cdot V^T)$; (do Gram-Schmidt on product)
- $A = URV$! R's diagonal entries are all 0 beyond the rank, all non-zero before, *with probability 1.*
- What about in floating-point, with *numerical rank*?

Randomized URV

- Numerical rank r: $\sigma_1 \geq \ldots \geq \sigma_r \geq \sigma_{r+1} \ldots \geq \sigma_n$;
  $\sigma_1/\sigma_r = O(1)$, $\sigma_r \gg \sigma_{r+1}$.

- The same algorithm works for *numerical* rank with high probability, assumming an appropriately large gap $\sigma_r \gg \sigma_{r+1}$.

- Idea existed before, e.g., Rokhlin et al., Mahoney et al., Tropp et al., for $r = o(n)$. Here $r = \Theta(n)$ is necessary, for Divide and Conquer.

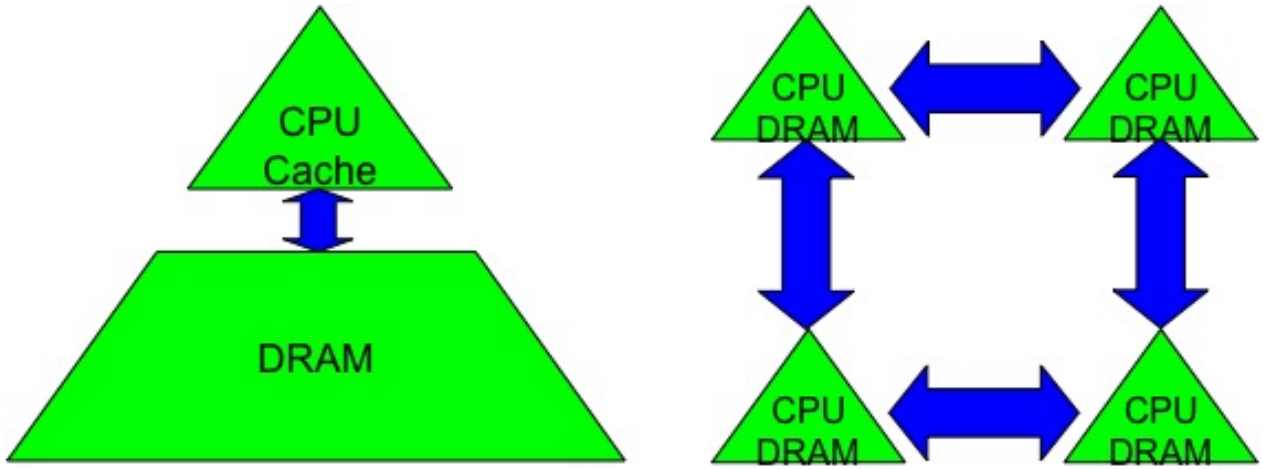
- Proof uses the order of smallest eigenvalue for a $\beta = 1, 2$ Jacobi matrix (D., 2010).

- *The resulting algorithm is easily parallelizable and also minimizes communication.*

Communication Cost Model

Algorithms have two costs:

1. arithmetic (flops)
2. communication: moving data between
   - levels of a memory hierarchy (sequential case)
   - processors over a network (parallel case)



Communication Cost Model

- Running time of an algorithm is sum of 3 terms:
  - # flops * time per flop
  - # words moved / bandwidth
  - # messages * latency

Communication Cost Model

- Exponentially growing gaps between
  - Sequentially:
    time per flop $\ll$ 1 / network BW $\ll$ network latency
    improving 59% per year vs. 26% per year vs. 15% per year
  - In parallel:
    time per flop $\ll$ 1 / memory BW $\ll$ memory latency
    improving 59% per year vs. 23% per year vs. 5.5% per year
- Need to reorganize linear algebra to *avoid* communication (# words and # messages moved)

Eigenvalue Algorithms on Sequential Machines

- Communication lower bounds

$$\#\text{ words} = \Omega\left(\frac{n^3}{\sqrt{M}}\right) \qquad \#\text{ messages} = \Omega\left(\frac{n^3}{M^{3/2}}\right)$$

- Factors by which algorithms exceed lower bounds:

	New Algorithms		*LAPACK*	
Algorithm	# words	# messages	# words	# messages
Symm Eig/SVD	$O(1)$	$O(1)$	$O\left(\sqrt{M}\right)$	$O\left(\sqrt{M}\right)$
Nonsymm Eig	Optimal!			

- M is fast memory size, n is problem size
- New algorithms have same communication complexity as QR decomposition and achieve lower bound

Eigenvalue Algorithms on Parallel Machines

- Communication lower bounds

$$\#\text{ words} = \Omega\left(\frac{n^2}{\sqrt{P}}\right) \qquad \#\text{ messages} = \Omega\left(\sqrt{P}\right)$$

- Factors by which algorithms exceed lower bounds:

	New Algorithms		*ScaLAPACK*	
Algorithm	# words	# messages	# words	# messages
Symm Eig/SVD	$O(\log P)$	$O(\log P)$	$O(\log P)$	$O\left(\frac{n}{\sqrt{P}}\right)$
Nonsymm Eig	Nearly optimal!		$O\left(\sqrt{P}\log P\right)$	$O(n \log P)$

- P is number of processors, n is problem size
- New algorithms have same communication complexity as QR decomposition and achieve lower bound (up to polylog factors)

History of Spectral Divide and Conquer

- Ideas go back to Bulgakov, Godunov, Malyshev ('88-'89)
- Bai, Demmel, Gu '97
  - reduced to matmul, QR, generalized QR with pivoting
- Demmel, D., Holtz, '07
  - instead of QR with pivoting, use RURV (randomized URV)
  - requires matmul and QR, no column pivoting
- Demmel, Grigori, Hoemmen, Langou '08
  - communication-optimal QR decomposition ("CAQR")
- (tangential) Ballard, Demmel, Holtz, Schwartz '09-'10
  - communication lower bounds on LA
  - communication-optimal Cholesky

Overview of (Ballard, D., Demmel) Algorithm '10

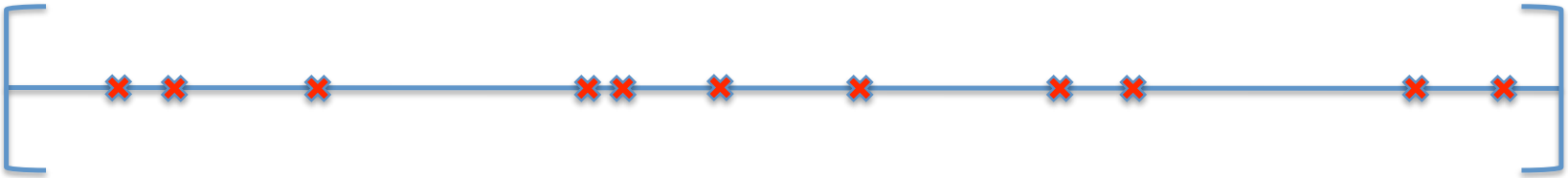
One step of divide and conquer:

1. Compute $\left(I + (A^{-1})^{2^k}\right)^{-1}$ implicitly
   - maps eigenvalues of A to 0 and 1 (roughly)
2. Compute rank-revealing decomposition to find invariant subspace
3. Output block-triangular matrix

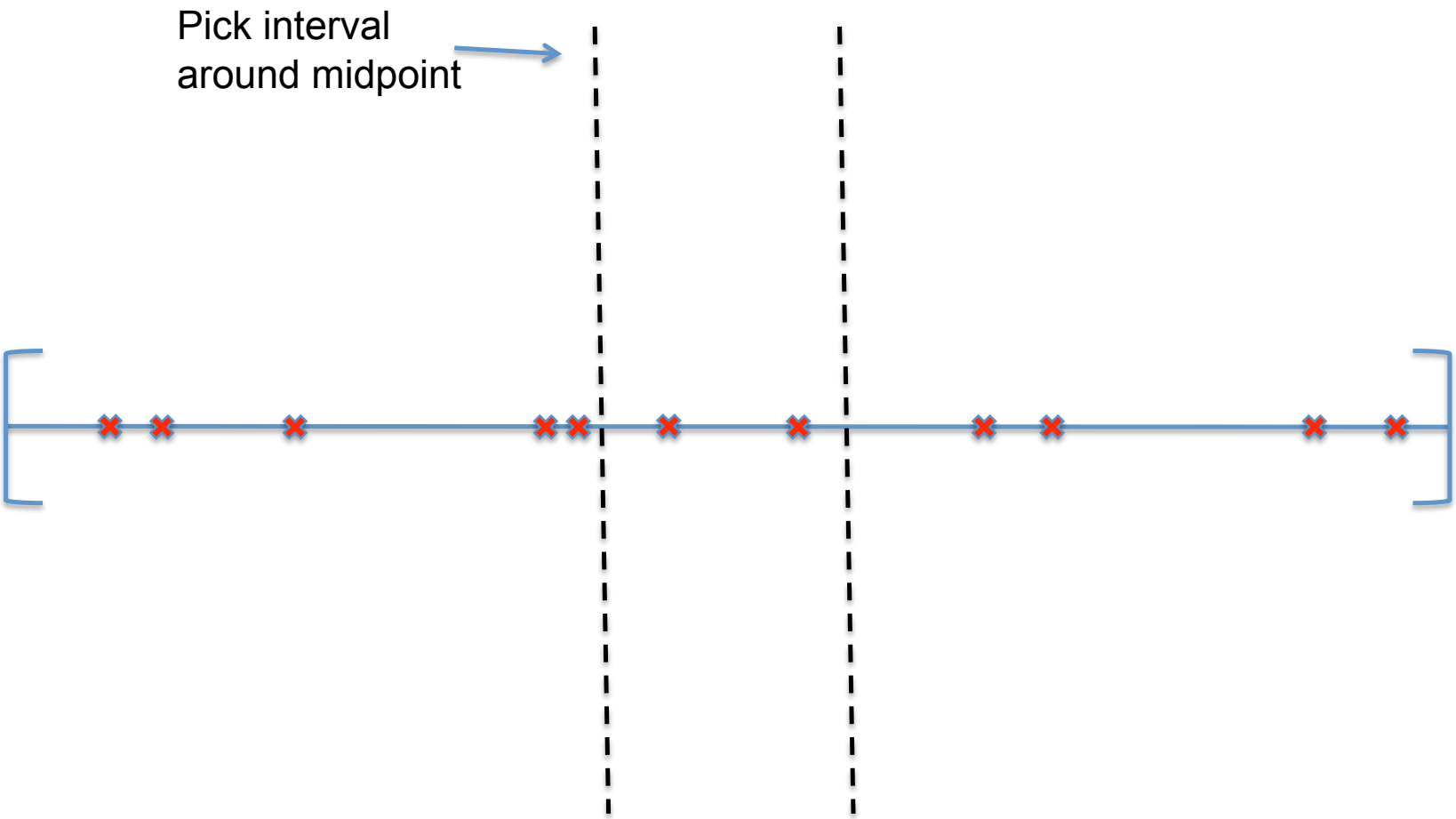
$$A_{\text{new}} = U^* A U = \begin{bmatrix} A_{11} & A_{12} \\ \varepsilon & A_{22} \end{bmatrix}$$

- block sizes chosen to minimize norm of ε
- eigenvalues of A_{11} all lie outside unit circle, eigenvalues of A_{22} lie inside unit circle, subproblems solved recursively

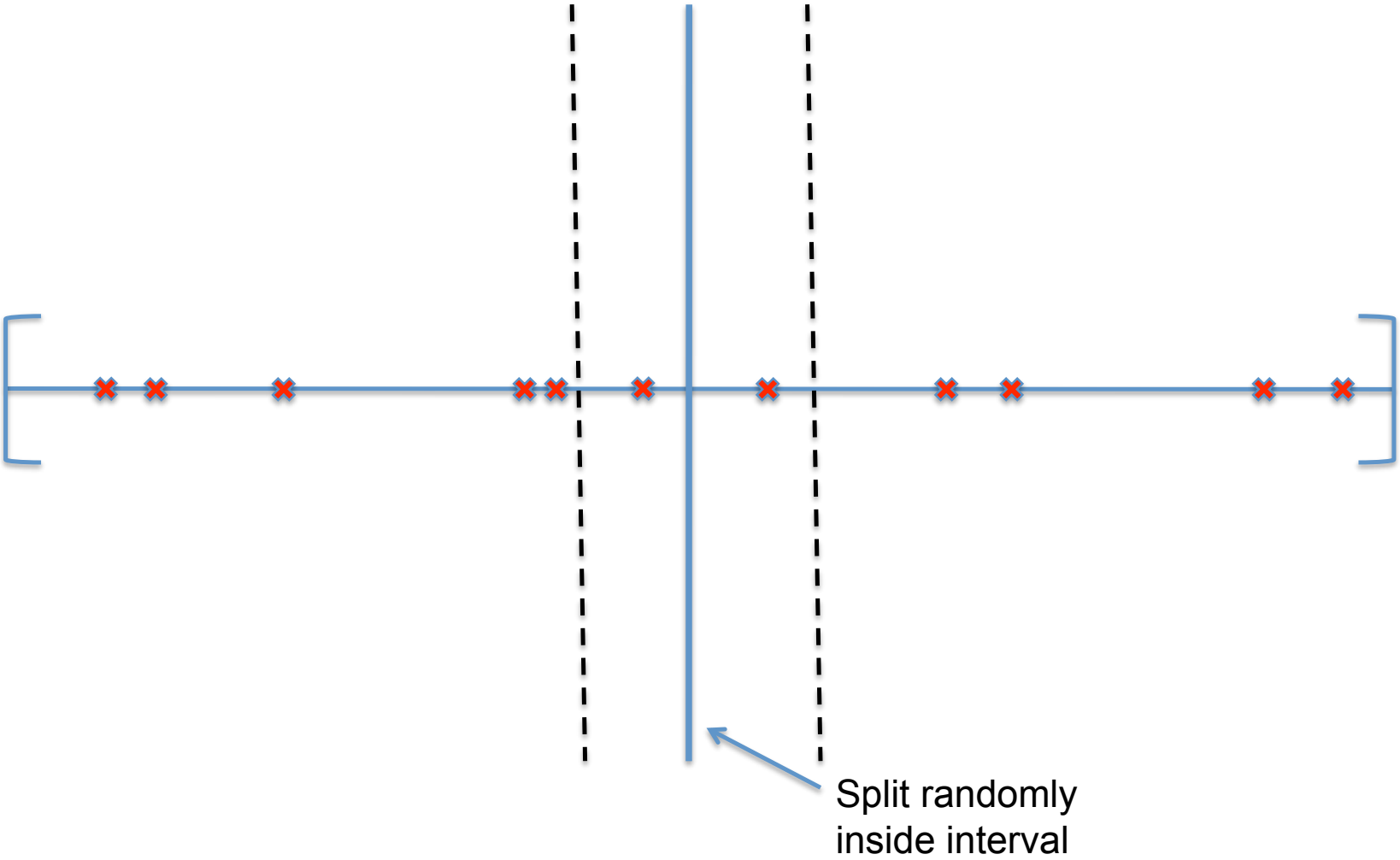
Randomized Bisection, symmetric case



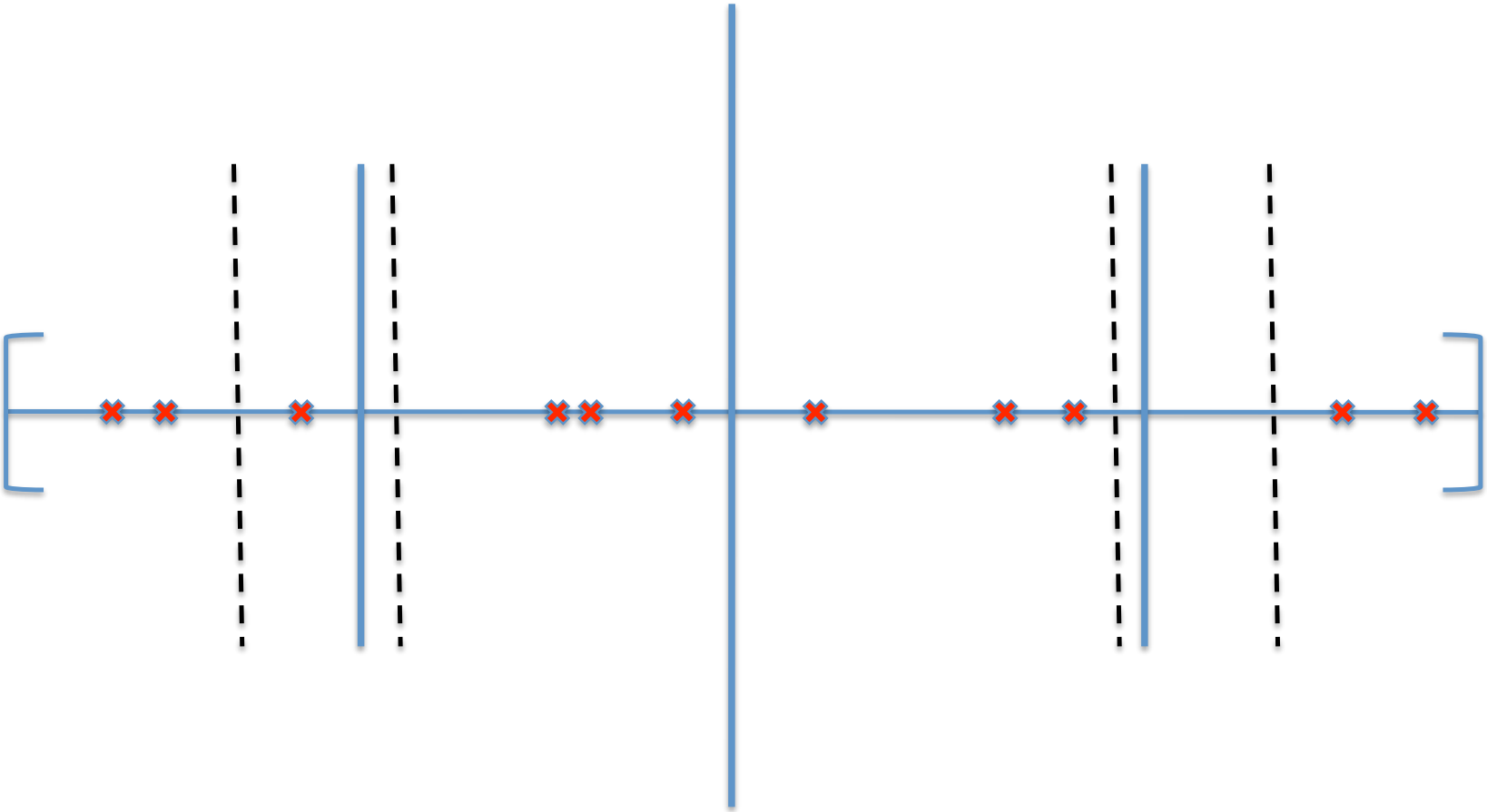
Randomized Bisection, symmetric case



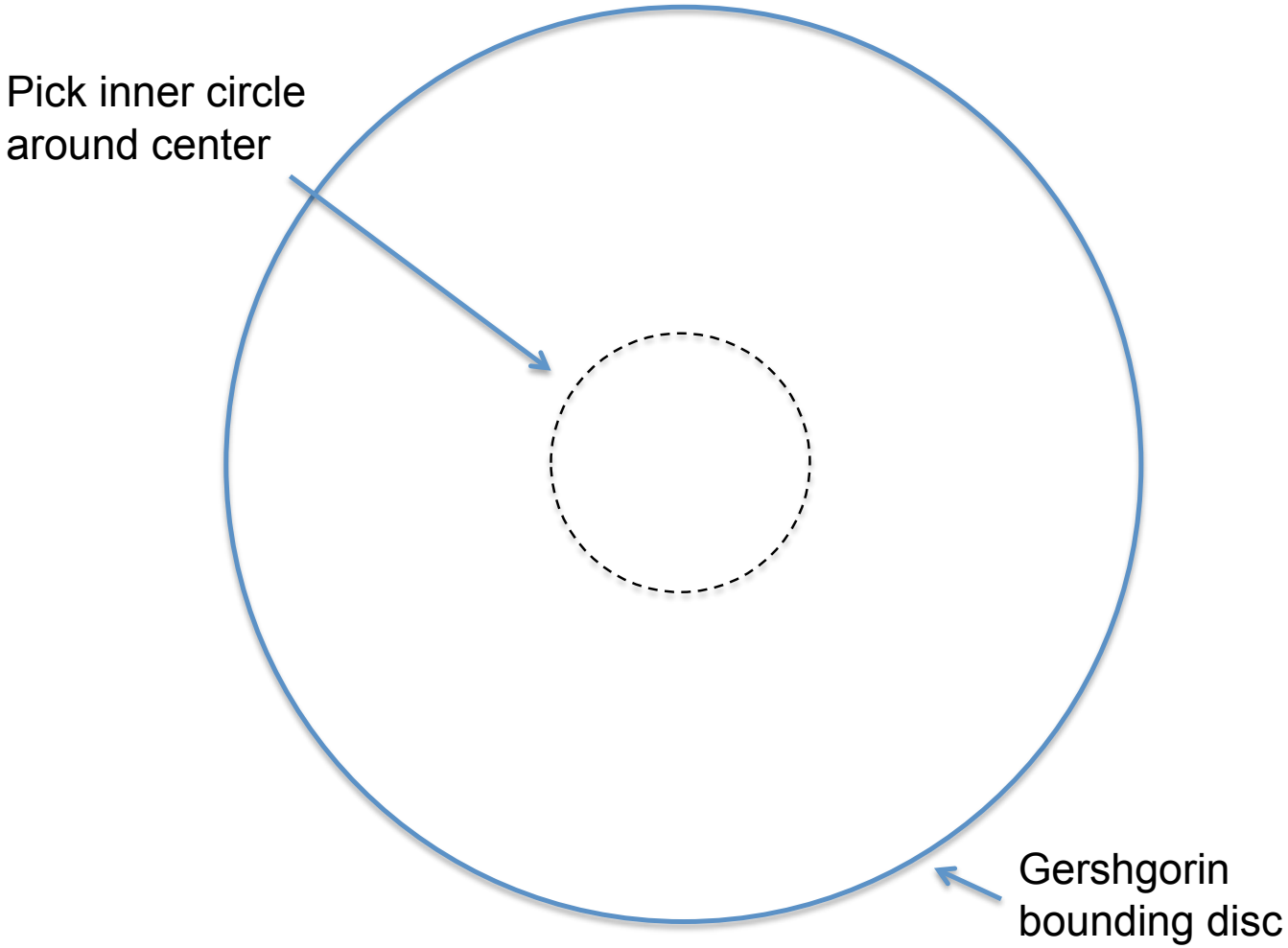
Randomized Bisection, symmetric case



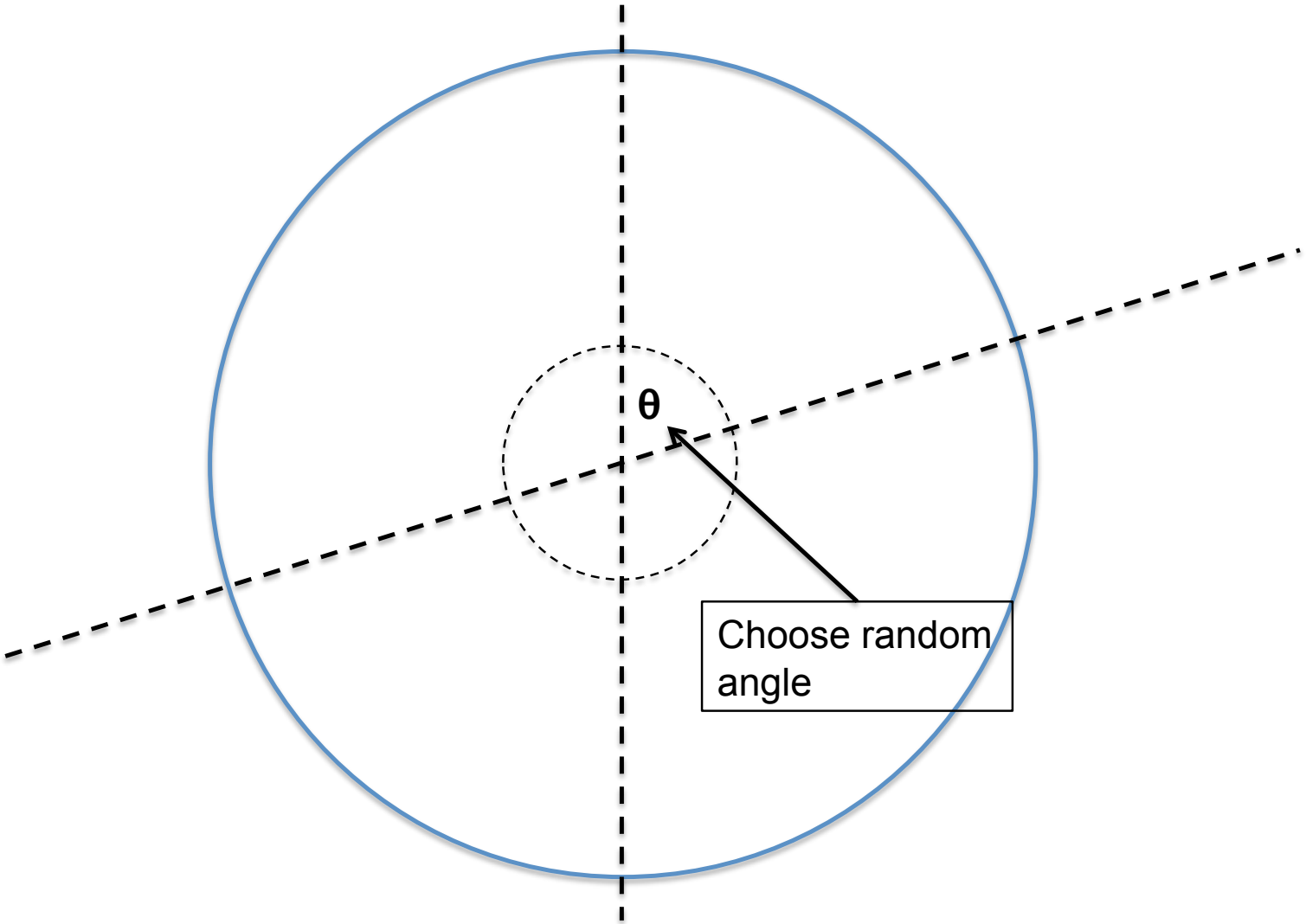
Randomized Bisection, symmetric case



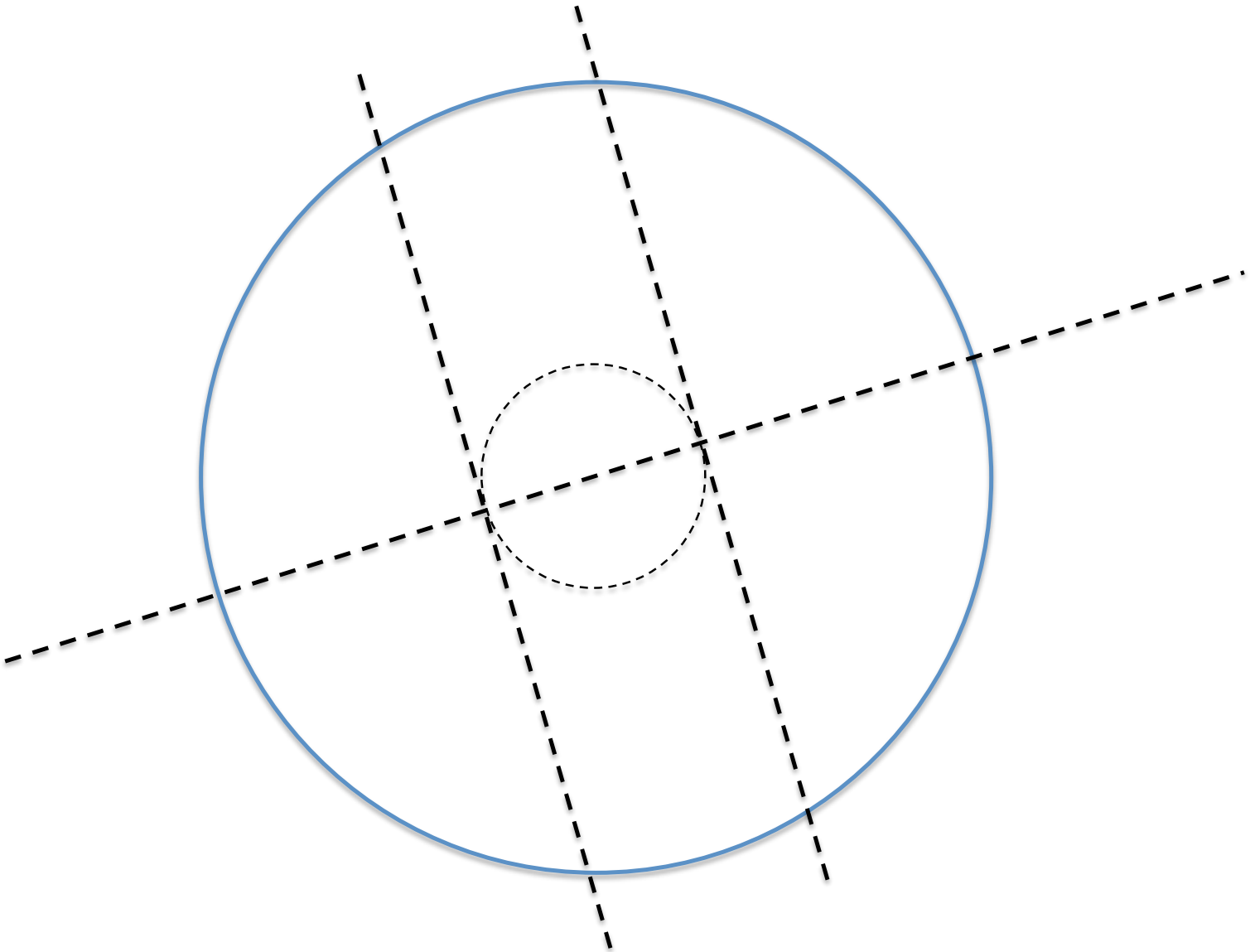
Randomized Bisection, non-symmetric case



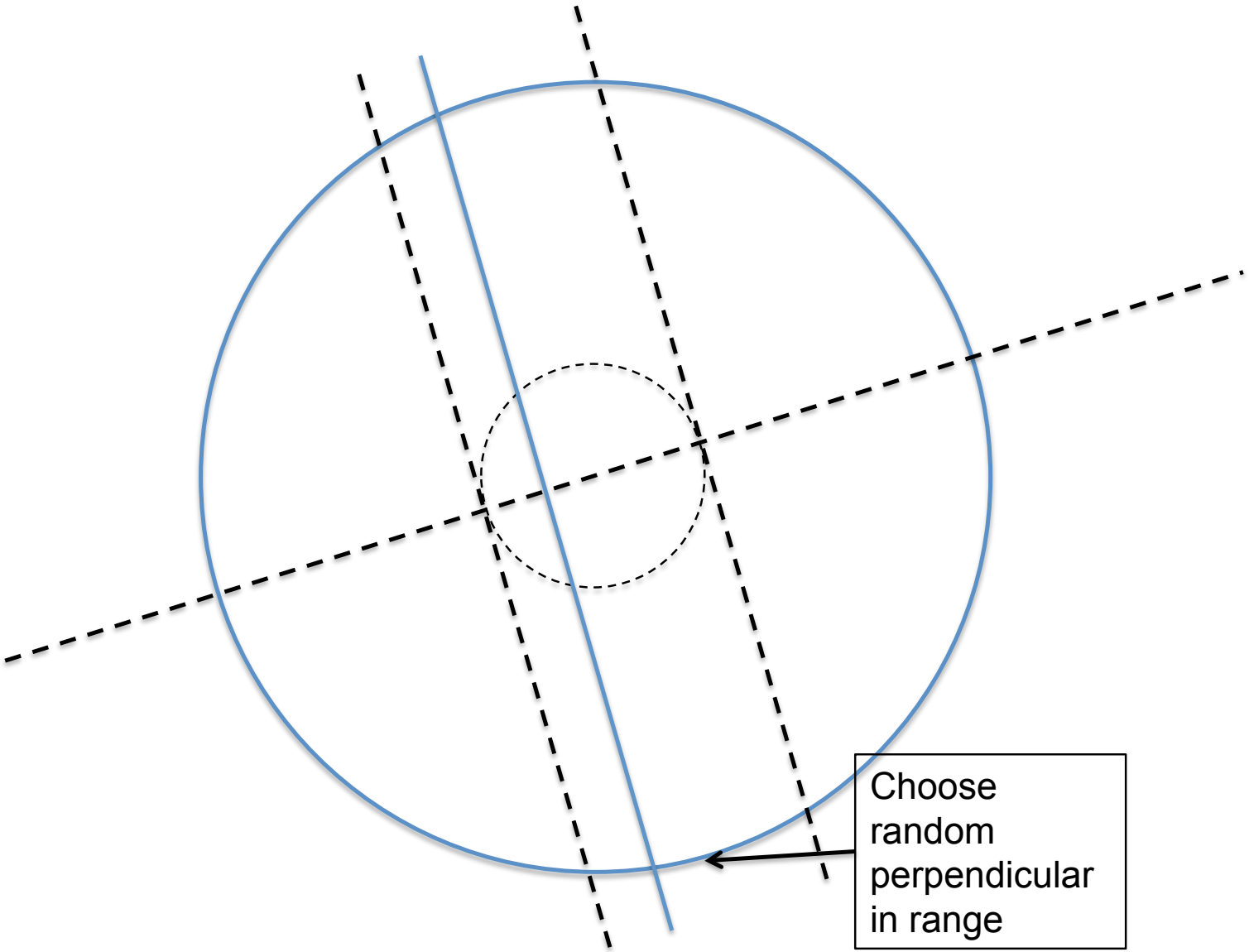
Randomized Bisection, non-symmetric case



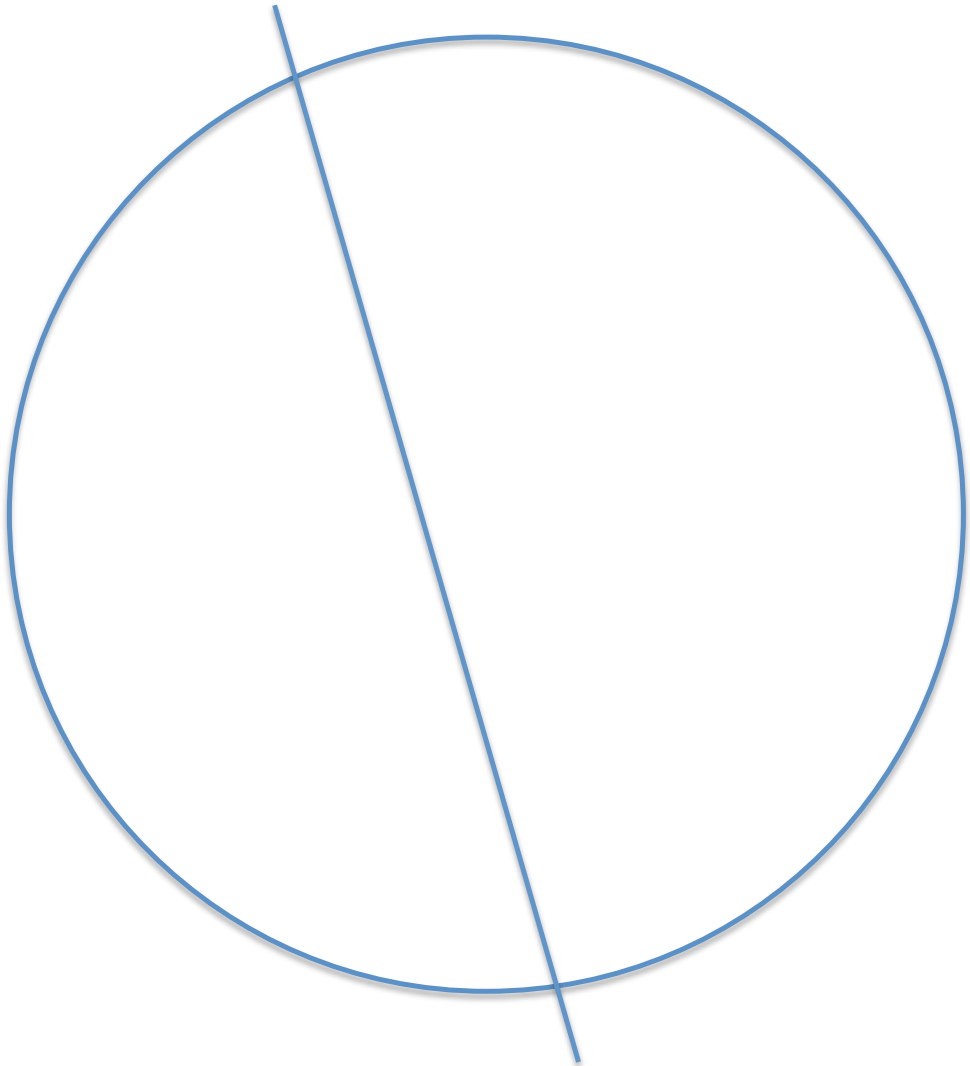
Randomized Bisection, non-symmetric case



Randomized Bisection, non-symmetric case



Randomized Bisection, non-symmetric case



Communication Upper Bound (sequential case)

- M = memory size, γ = cost of flop, β = inverse bandwidth, α = latency

Cost of one step of divide-and-conquer is

$$C_{\text{D+C}}(n) = \alpha \cdot O\left(\frac{n^3}{M^{3/2}}\right) + \beta \cdot O\left(\frac{n^3}{\sqrt{M}}\right) + \gamma \cdot O(n^3)$$

Assuming we split the spectrum by some fraction $1 > f > 0$ each time (likely b/c of randomization!): total cost of the entire algorithm is asymptotically the same!

Algorithm attains lower bound.

Communication Upper Bound (parallel case)

- P = # processors, γ = cost of flop, β = inverse bandwidth, α = latency

Cost of one step of divide-and-conquer is

$$C_{\text{D+C}}(n,P) = \alpha \cdot O\left(\sqrt{P}\log P\right) + \beta \cdot O\left(\frac{n^2}{\sqrt{P}}\log P\right) + \gamma \cdot O\left(\frac{n^3}{P}\log P\right)$$

Subproblems can be solved in parallel. Assuming we split the spectrum by some fraction $1 > f > 0$ each time (likely b/c of randomization!): total cost of the entire algorithm is asymptotically the same!

Algorithm attains lower bound (to within logarithmic factors).

What to take home

- If one can reach the complexity exponent of matrix multiplication, one can do all linear algebra *fast and stably*.
- The above works for **large** matrices (constants may be large).
- *Randomizing* can mean *stabilizing*.

- *Divide and Conquer* is a very good strategy (for EIG/SVD).
- Communication is bad and must be avoided, even if it means doing more work.