Lecture 11: Finishing Locks; Vectors

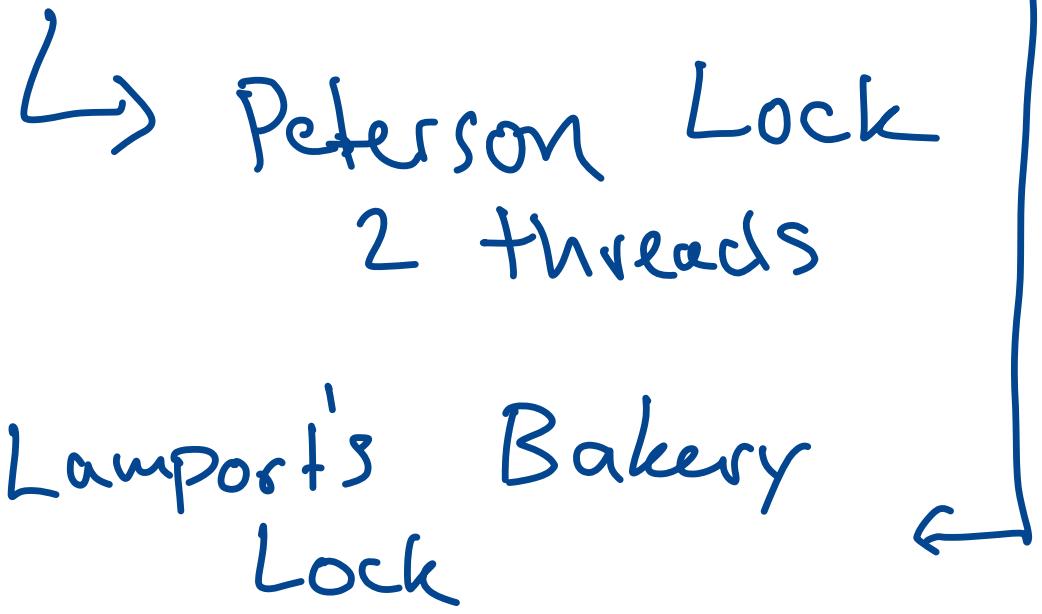
COSC 273: Parallel and Distributed Computing

Spring 2023

HW 02 Posted today

↪ Due Friday

Last Time: Fair Locks, More Threads



array size = # threads

thread IDs 0, 1, 2, ... size

Lamport's Bakery Algorithm

critical section

Fields:

- `boolean[] flag`
  - `flag[i] == true` indicates `i` would like enter CS
- `int[] label`
  - `label[i]` indicates "ticket" number held by `i`

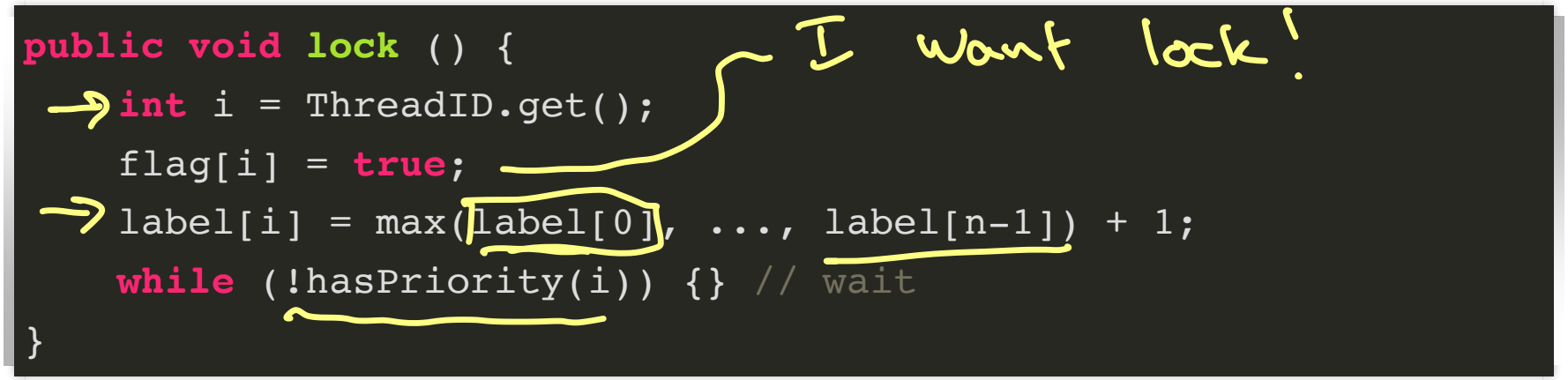
Initialization:

- set all `flag[i] = false`, `label[i] = 0`

Locking

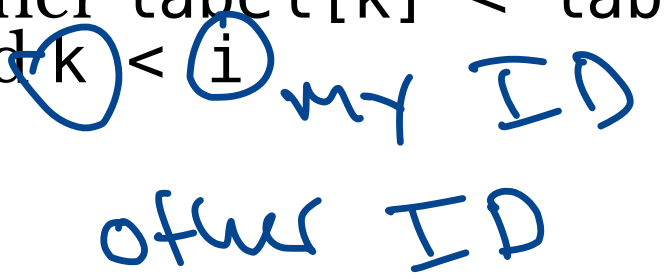
Locking Method:

```
public void lock () {
  int i = ThreadID.get();
    flag[i] = true;
  label[i] = max(label[0], ..., label[n-1]) + 1;
    while (!hasPriority(i)) {} // wait
}
```



The method `hasPriority(i)` returns `true` if and only if there is no `k` such that

- `flag[k] == true` and
- either `label[k] < label[i]` or `label[k] == label[i]` and `k < i`



Unlocking

Just lower your flag:

```
public void unlock() {
    flag[ThreadID.get()] = false;
}
```

Bakery Algorithm is Deadlock-Free

```
public void lock () {
    int i = ThreadID.get();
    flag[i] = true;
    label[i] = max(label[0], ..., label[n-1]) + 1;
    while (hasPriority(i)) {} // wait
}
```

never caught stalemate

Why?

Obs. Some thread has priority at all times, that thread obtains lock

- among flag[i] == true
- smallest label, ties by ID.

if someone wants lock, someone gets it

First-come-first-served (FCFS)

- If: *A* writes to `label` before *B* calls `lock()`,
- Then: *A* enters CS before *B*.

```
public void lock () {
    int i = ThreadID.get();
    flag[i] = true;
    label[i] = max(label[0], ..., label[n-1]) + 1;
    while (!hasPriority(i)) {} // wait
}
```

Why?

- A writes label
- B calls lock
  - writes label

=> B has lower priority b/c B's label ≥ 1 more than A's

Bakery Algorithm is Starvation-Free

Why?

Show: If I call lock, I eventually acquire lock

If: not highest priority, then highest P gets lock by D.F. + F.C.F.S.

→ this thread will never again have higher priority after they release lock + before I get lock (FCFS)

→ new fewer higher priority threads, repeat arg. for next highest.

Bakery Algorithm is Starvation-Free

Why?

Thread `i` calls `lock()`:

- `i` writes `label[i]`
- By FCFS, subsequent calls to `lock()` by `j != i` have lower priority
- By deadlock-freedom every `k` ahead of `i` eventually releases lock

So:

- `i` eventually served

Bakery Algorithm Satisfies MutEx

```
public void lock () {
    int i = ThreadID.get();
    flag[i] = true;
    label[i] = max(label[0], ..., label[n-1]) + 1;
    while (!hasPriority(i)) {} // wait
}
```

Suppose not:

- A and B concurrently in CS
- Assume $(\text{label}(A), A) < (\text{label}(B), B)$ ← when in CS

Proof (Continued)

Since B entered CS:

- Must have read
  1. $(\text{label}(B), B) < (\text{label}(A), A)$, or
  2. $\text{flag}[A] == \text{false}$

Why can't 1 happen?

when B read A's label?

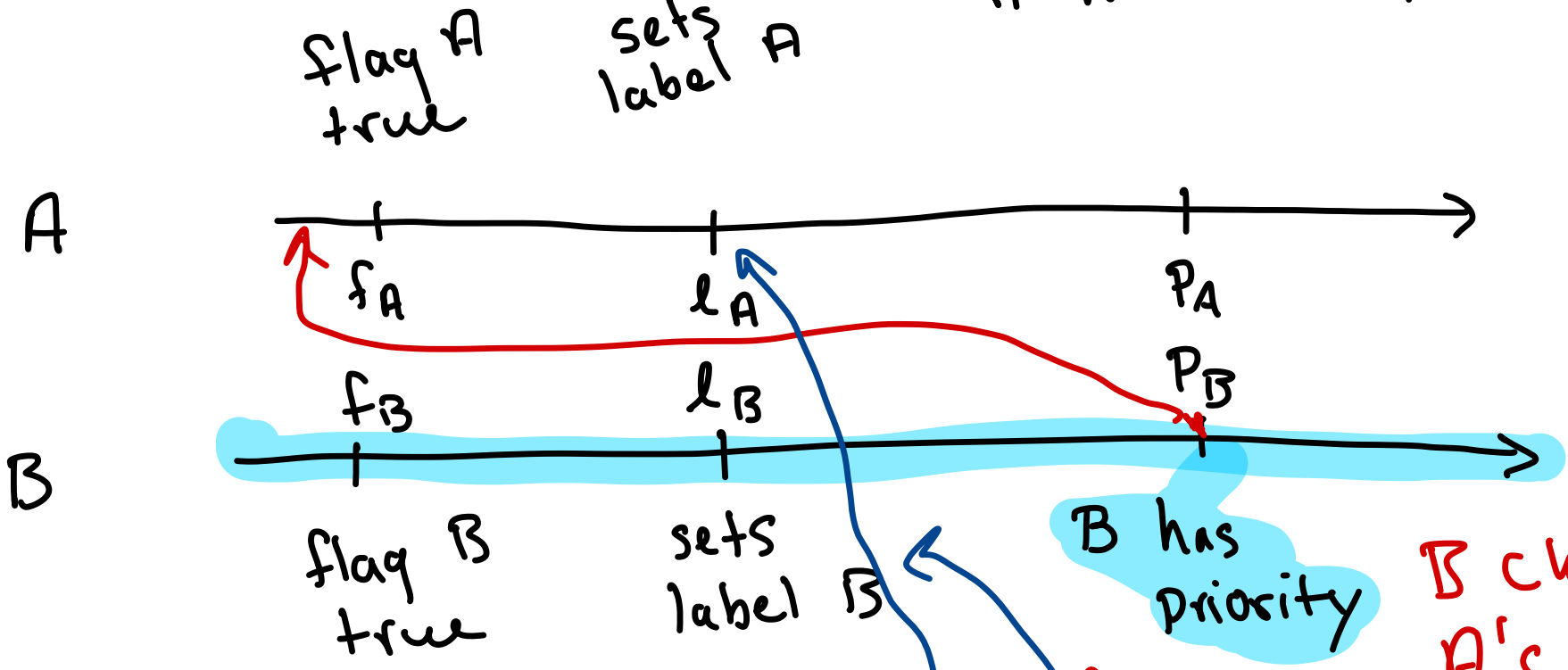
$\Rightarrow$ A's priority increased from when B read A's priority to when B entered crit. sec.

But priorities only decrease!

Why can B not have read flag[A] == false?

Compare Timelines!

A has priority



If B reads flag[A] == false

$\Rightarrow P_B < f_B$ set here

$\Rightarrow$ label[A] > label[B] set here

$\Rightarrow$ contradicts assumption that A had higher priority $\neq$

Conclusion?

Lamport's Bakery Algorithm:

1. Works for any number of threads
2. Satisfies MutEx and starvation-freedom

Is the bakery algorithm practical?

Two Issues:

1. For n threads, need arrays of size n
   - `hasPriority` method is costly
   - what if we don't know how many threads?
2. Assume threads have sequential IDs `0, 1,...`
   - not the case with Java!
   - thread IDs are essentially random `long` values

Homework 2 will have questions that address these issues.

Remarkably

We cannot do better!

- If *n* threads want to achieve mutual exclusion + deadlock-freedom, must have *n* read/write registers (variables)

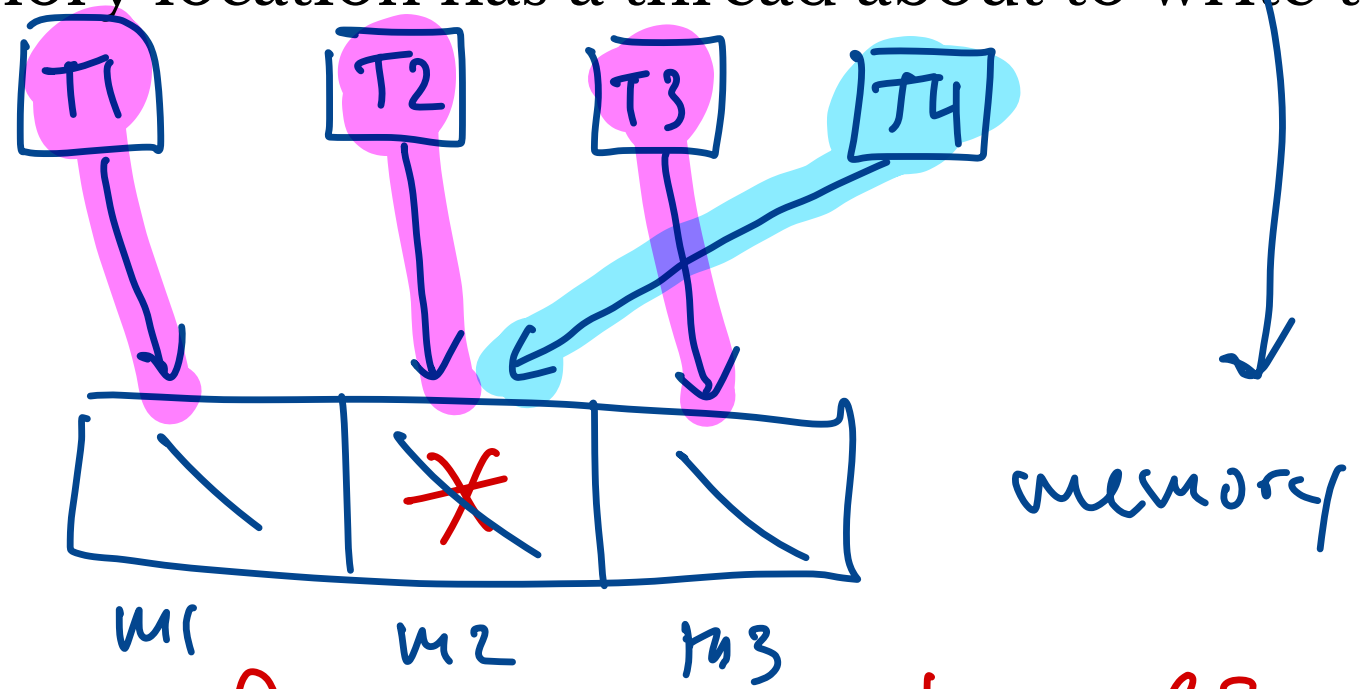
Lower Bound Argument Sketch

Consider n threads, $m < n$ shared memory locations

- fix some mutex protocol

A **covering state** is a step in an execution in which:

1. Each thread's next step is a `write` operation
2. Each thread's view is consistent with CS unoccupied
3. Each memory location has a thread about to write to it



BY Deadlock-freedom, T4 enters CS
1 step each for T1, T2, T3

Claim

If an execution reaches a covering state, then the protocol does not satisfy mutual exclusion.

Why?

Finishing Lower Bound Argument

Show. Any protocol with $m < n$ memory locations attains a covering state in some execution.

- Read *AMP* Section 2.9 for details

Finishing Lower Bound Argument

Show. Any protocol with $m < n$ memory locations attains a covering state in some execution.

- Read *AMP* Section 2.9 for details

Consequences:

- **If** only *synchronization primitives* are `read`/`write` then n shared memory locations are necessary for deadlock-free mutual exclusion with n threads
  - Bakery algorithm is nearly optimal (memory of $2n$)
- Led to development of **stronger** primitives

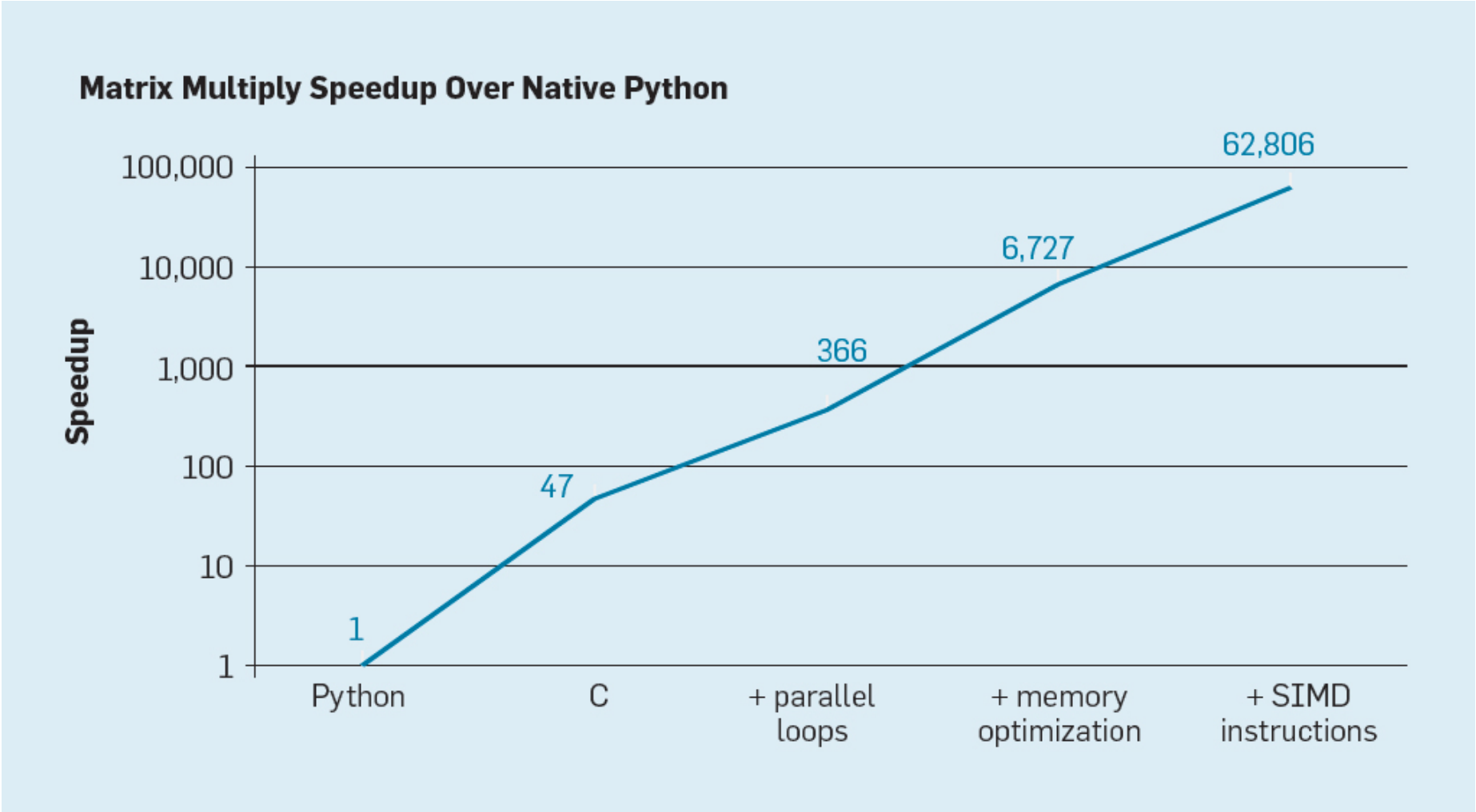
A Way Around the Bound

- Argument relies crucially on fact that the *only* atomic operations are `read` and `write`
- Modern computers offer more powerful atomic operations
- In Java, `AtomicInteger` class
  - `getAndIncrement()` is supported **atomic** operation

Homework 2 Use `AtomicIntegers` to get a cleaner and more efficient realization of Lamport's bakery idea.

Changing Gears

Performance, Again



More Powerful Hardware

In `Java`, `int` and `float` values are 32 bits long

In modern CPUs, registers are larger

- my computer: 256 bit registers

Naive Operations

```
int a = 573842;
int b = 3847253;
int c = a + b;
```

SIMD Parallel Operations

```
int a1 = 573842;
int b1 = 3847253;
int c1 = a1 + b1;

int a2 = 38657548;
int b2 = 438573;
int c2 = a2 + b2;
```

Naive Loops

```
int[] a = new int[n];
int[] b = new int[n];
int[] c = new int[n];

for (int i = 0; i < n; i++) {
   c[i] = a[i] + b[i];
}
```

Using Full Power

Suppose we can load `step` values into each register

```
int[] a = new int[n];
int[] b = new int[n];
int[] c = new int[n];

for (int i = 0; i < n; i += step) {
   c[i] = a[i] + b[i];
   c[i+1] = a[i+1] + b[i+1];
   ...
   c[i+step-1] = a[i+step-1] + b[i+step-1]
}
```

Java Vector API

Allows us to specify `Vector` objects

- `Vector` is like fixed-size array
- tune `Vector` (bit) size to same as hardware registers
- perform elementary operations on entire vectors

Java Vector API

Allows us to specify `Vector` objects

- `Vector` is like fixed-size array
- tune `Vector` (bit) size to same as hardware registers
- perform elementary operations on entire vectors

Notes:

- Vector API in Java 19, available as “incubator”
- Many optimizations already done (without `Vector`)

Example

Find entry-wise minimum of arrays:

```
VectorSpecies<Float> SPECIES = FloatVector.SPECIES_PREFERRED;
    ...
public static float[] vectorMax(float[] a, float[] b) {
    float[] c = new float[a.length];
    int step = SPECIES.length();
    int bound = SPECIES.loopBound(a.length);
    ...
}
```

Example Continued

Find entry-wise minimum of arrays:

```
...
    int i = 0;
    for (; i < bound; i += step) {
        var va = FloatVector.fromArray(SPECIES, a, i);
        var vb = FloatVector.fromArray(SPECIES, b, i);
        var vc = va.max(vb);
        vc.intoArray(c, i);
    }
    for (; i < a.length; i++) {
        c[i] = Math.max(a[i], b[i]);
    }
    return c;
```

Speedup for Me

```
The FloatVector has 8 lanes.
Computing max array with simple methods...
That took 927 ms.
Computing max array with vector methods...
That took 572 ms.
The arrays are equal!
```

Next Lab

Use `Vector` operations to speed up programs!