
ORACLE®

ORACLE®

Data parallel programming and Java

John R. Rose
Oracle JVM Architect

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions.
The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

What is “data parallel” computing?

- Divide and conquer
  - Data is broken up into chunks (inputs, outputs, temps)
  - Each chunk is processed independently from the others
- Homogeneous computation
  - All the freedom is in the data, not the code
  - Operations are about the same everywhere
- Hardware requirements are modest
  - Easy to compile to vectorized loops (Haswell, etc.)
  - Easy to avoid branches; accepts deeply pipelined CPUs
  - Synergizes with array-wise prefetch through memory fabric
  - Also works fine on “Single Instruction Multiple Data” hardware

Running example: sum of C ← A + B

- In classic Java:

```
for (int i = 0; i < A.length; i++)
  C[i] = A[i] + B[i];
```

- Fortran 95 explicit loop:

```
FORALL (I = 1:N) C(I) = A(I) + B(I)
```

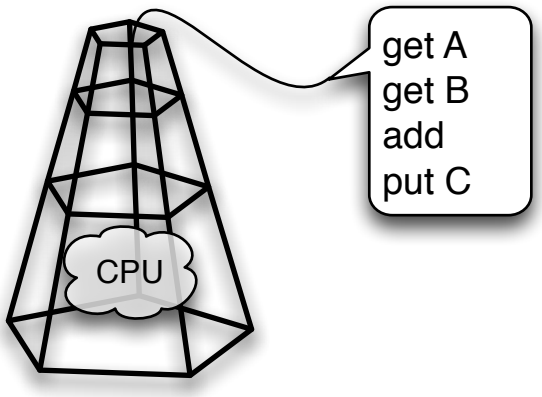
- Array languages (Fortran, R, Matlab, APL):

```
C "gets" A + B
```

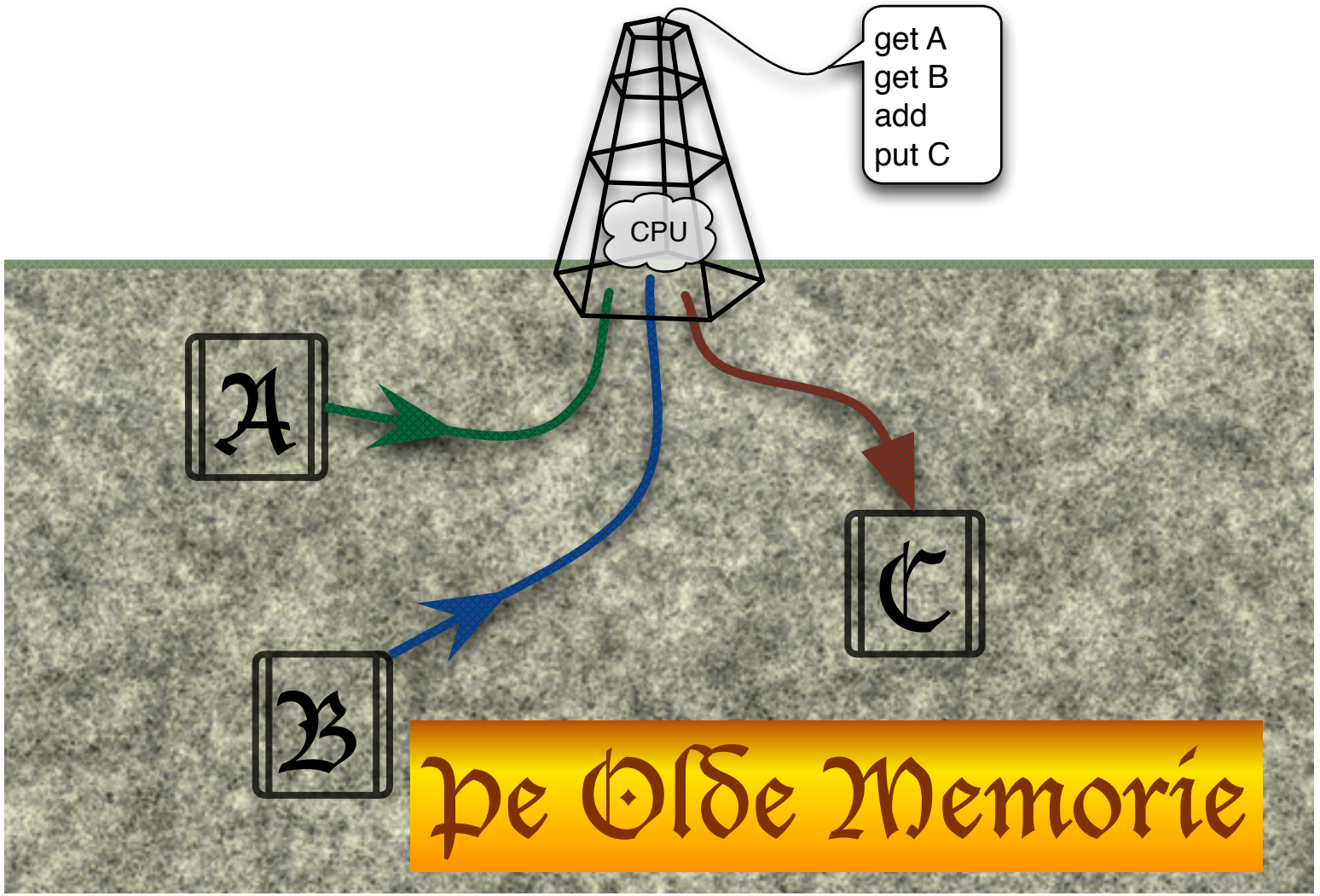
- In JDK 8 streams:

```
IntStream.range(0, A.length).parallel()
  .mapToDouble(i->A[i]+B[i])
    .toArray()
```

Can a coder dream of electric chips?



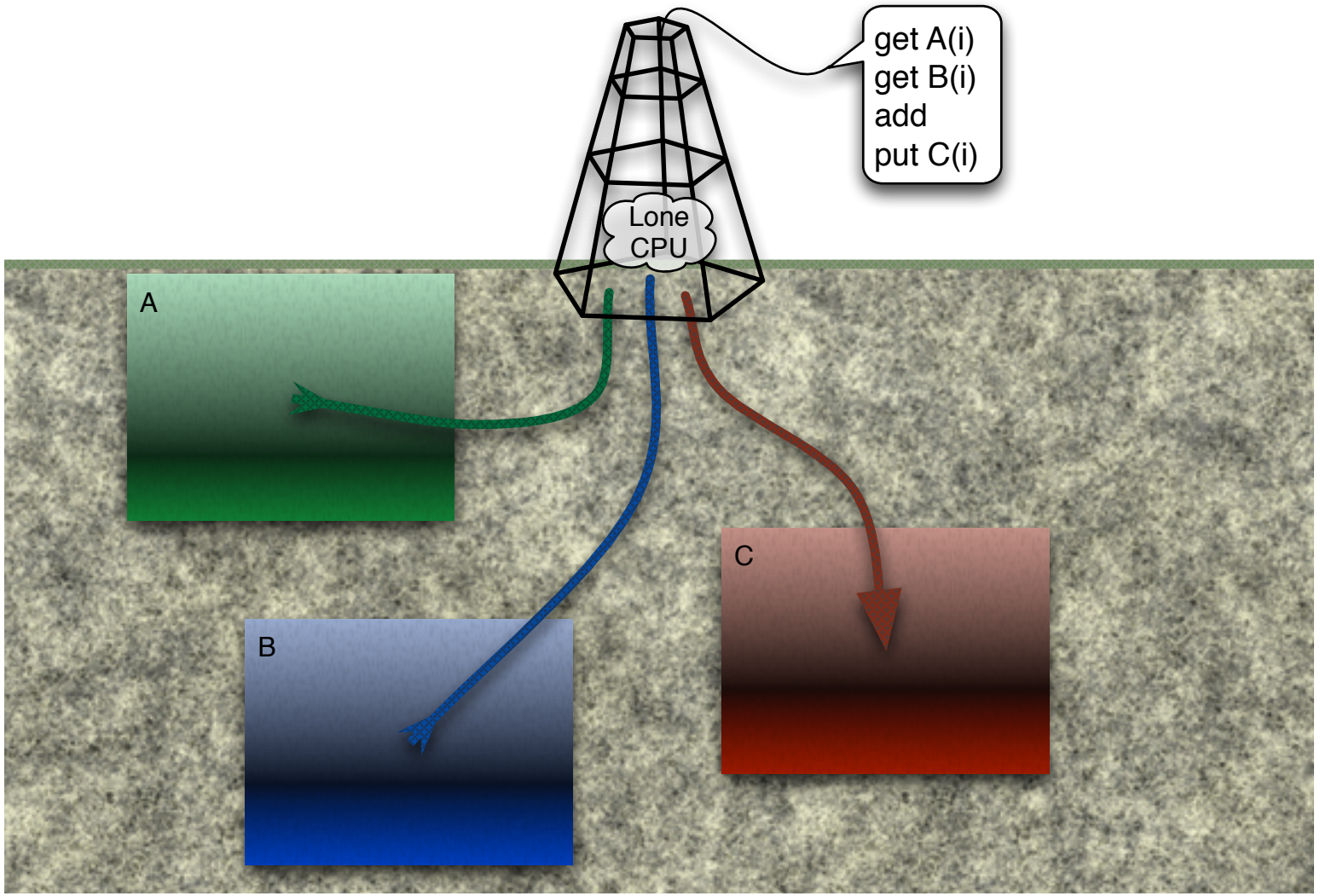
An ordinary load-store machine



working assumption: data = arrays

- little loss of generality to assume array-like data
  - linked lists or iterators are not sub-dividable
  - ArrayList is OK, of course, and so are buffers (NIO, etc.)
  - JDK 8 “spliterators” can be buffered in medium-sized arrays
- key properties of arrays and array-likes
  - subdivision is cheap (log or constant time)
  - streaming over subdivisions is cheap (linear with prefetch)
  - concurrent access to disjoint subdivisions is non-interfering
- classic Java arrays are OK but not exactly right
  - #include <arrays 2.0 talk>

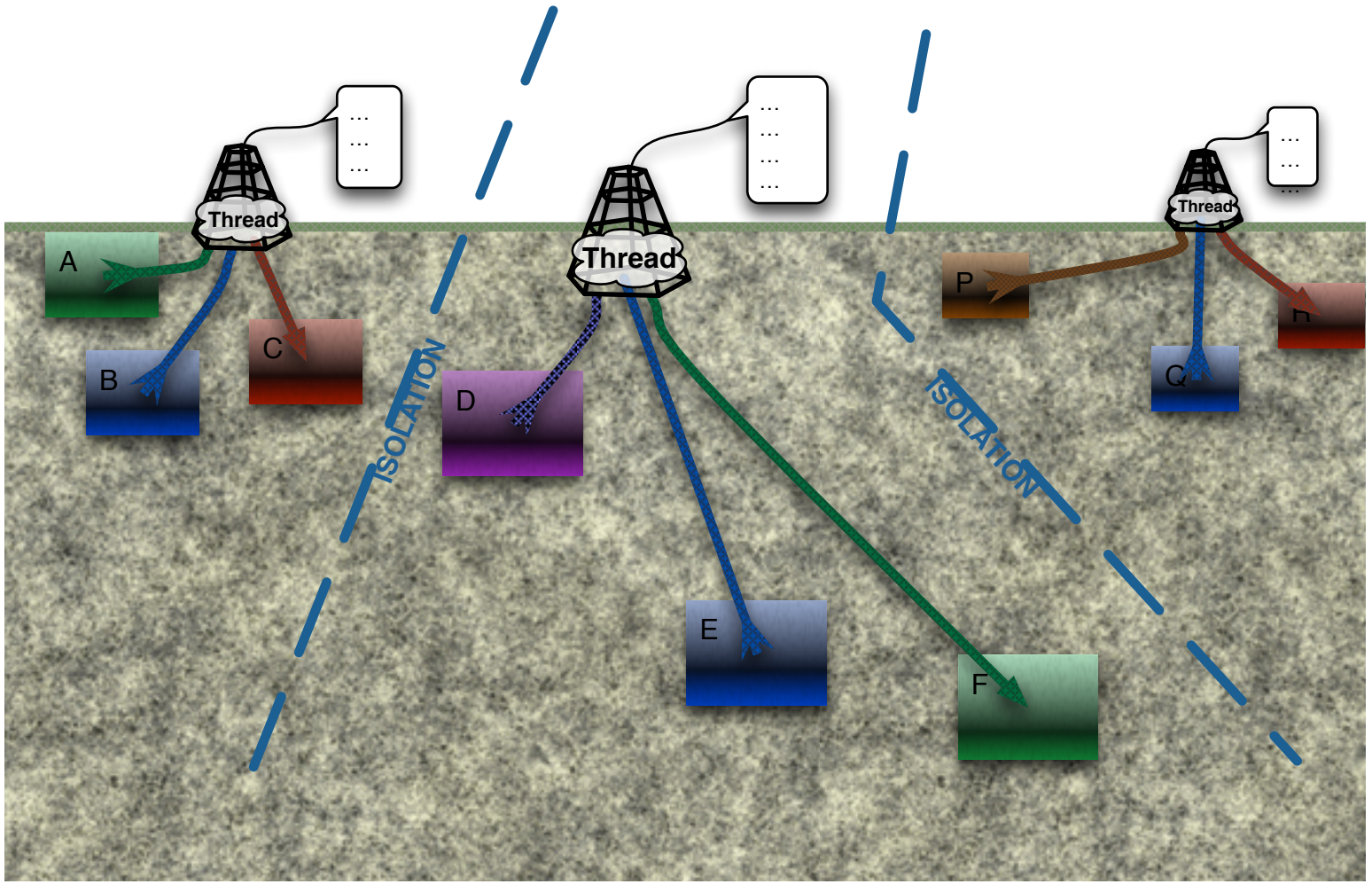
Load-store machine, with arrays



How the Java VM virtualizes a CPU

- a thread (usually more than one) containing:
  - a bytecode instruction stream
  - stack frames containing variables and other goodies
  - enough raw stack to make SOE rare in practice
- a big shared memory containing:
  - contains simple structs and arrays
  - self-identifying (reflectable) data and managed pointers
  - enough raw heap to make OOME rare in practice
- concurrency rules which:
  - reconcile local thread R/W actions with global state
  - provide ways to coordinate via synchronization events
  - give just enough *happens-before* relations to avoid bad races

So let's go multi-threaded!

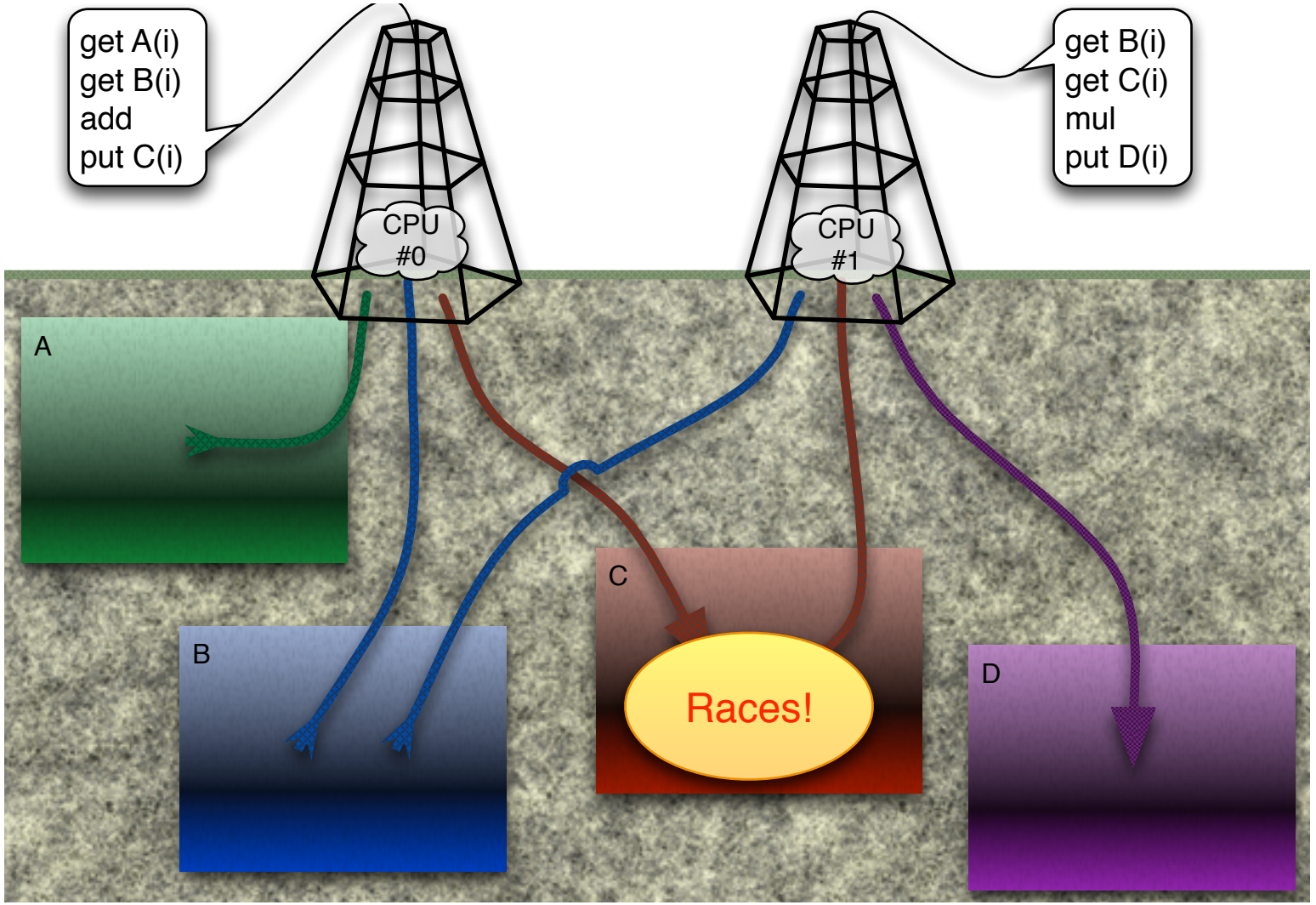


Old school multi-threading

- "MIMD" = Multiple Instruction Multiple Data
  - very general
  - works fine if thread effects are disjoint
  - mix in with mutable data everywhere for extra "generality"
- code must synchronize carefully
  - any other thread might be executing any other code
  - it is difficult to reason about all possible interleavings
- footprint from a big, rich `java.lang.Thread` object
  - requires a large pre-allocated control stack
  - comes with extra amenities like `java.lang.ThreadLocal`

ORACLE

Threads can step on each others' toes



What went wrong?

- The “Central” aspect of “CPU” causes bugs
  - the memory is central (if anything is); processing is not
- An output of CPU#0 was a racy input of CPU#1
  - Best practice: “Settle” and freeze input arrays before use.
  - Value Objects JEP includes a “freeze array” operator.
- Problems also arise if two CPUs use the same output
  - Best practice: Write each output variable exactly once.
  - Create outputs late and fully formed – `Stream.toArray`

BTW, what is right, despite?

- Bytecode semantics
  - valuable medium-level notation, workable in SIMD & MIMD
- Managed heap, concurrency model, safety/security
- Supported: objects, methods, lambdas, streams, etc.
- Both high-level APIs and low-level utilities

- Let's keep all that...

In search of the right notation

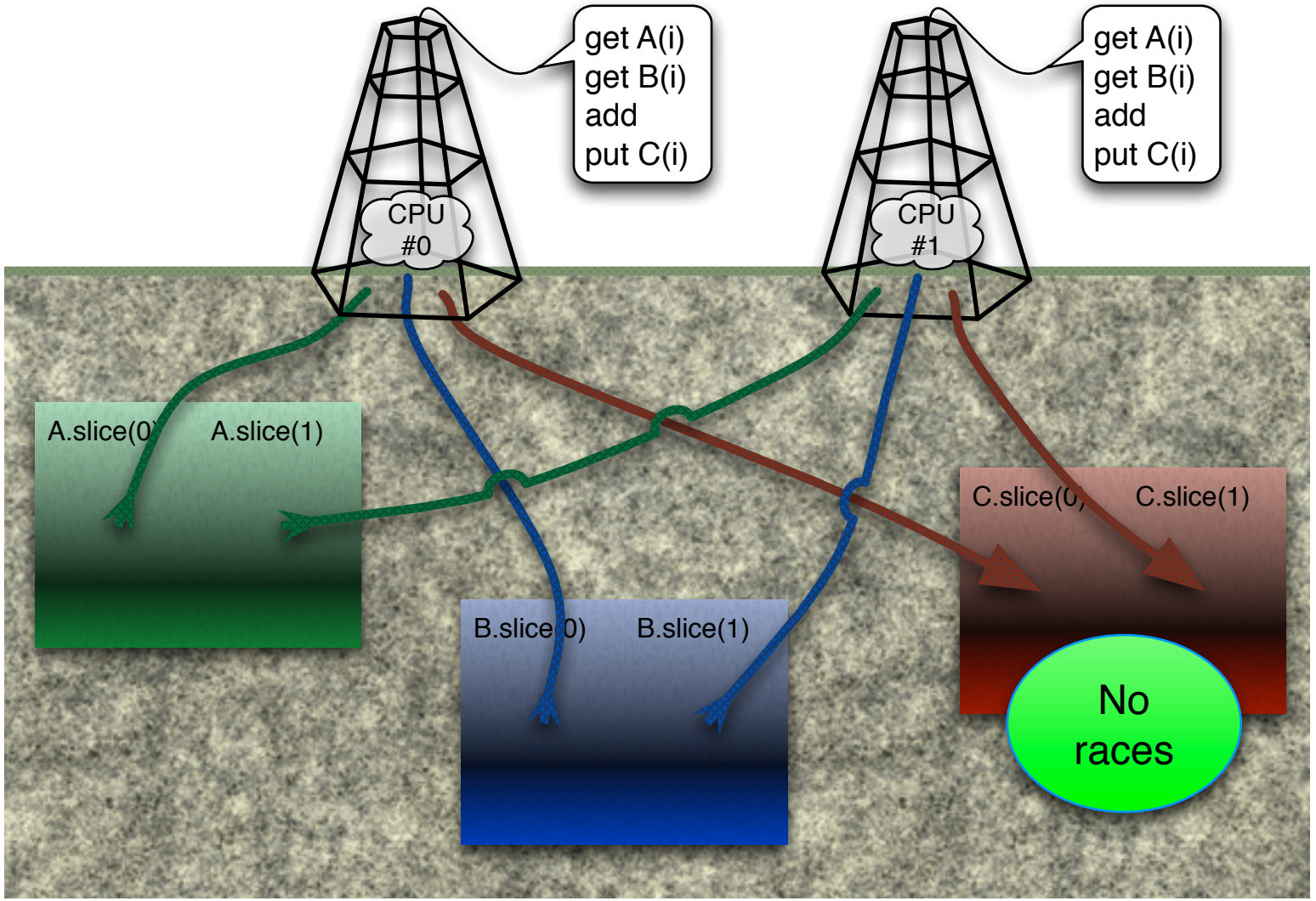
- MIMD is overly general for simple things like C=A+B
  - loops promise too much sequencing; also verbose
- For many end-users, streams give simple power
  - “just add dot-parallel” – reasonable recipe, JVM must honor
- Bulk array-oriented operations are friendly too
  - note deep popularity of R language
  - looping is implicitly supplied from array shape rules
  - room here, maybe, for a DSL in Java some day
  - for a quick win, consider injecting Arrays into T[] via defaults
- Low-level big array of small pre-placed structs
  - (a la *Lisp, C*; more later)

ORACLE

Timing can be everything

- User model and notation must explain sequencing
  - (at least to the extent that there are side effects underfoot)
- Can set concurrency rules various ways:
  - global clock (lockstep per bytecode, or hidden loops)
  - local clocks with explicit synch. handshakes
  - queues, dataflow? (at which scale?) – streams feel better
  - warps (thread swarms) – divergence can be a problem here
  - library framework schedules callbacks (GCD, FJ, streams)
- To avoid stepping on your partner's foot, try lockstep
  - (If you still stomp the foot, it will be much more predictable.)

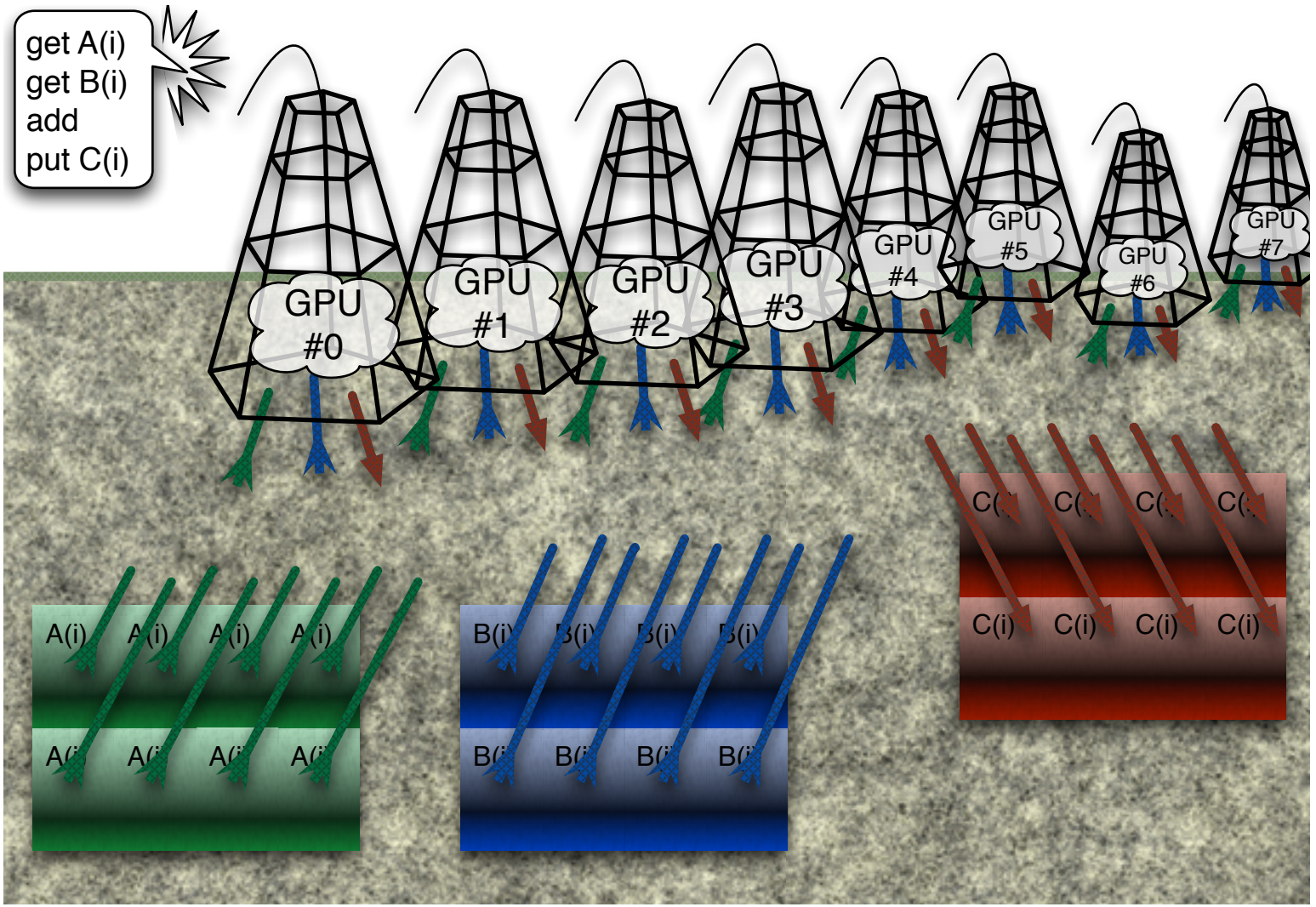
Partition the data, not the code



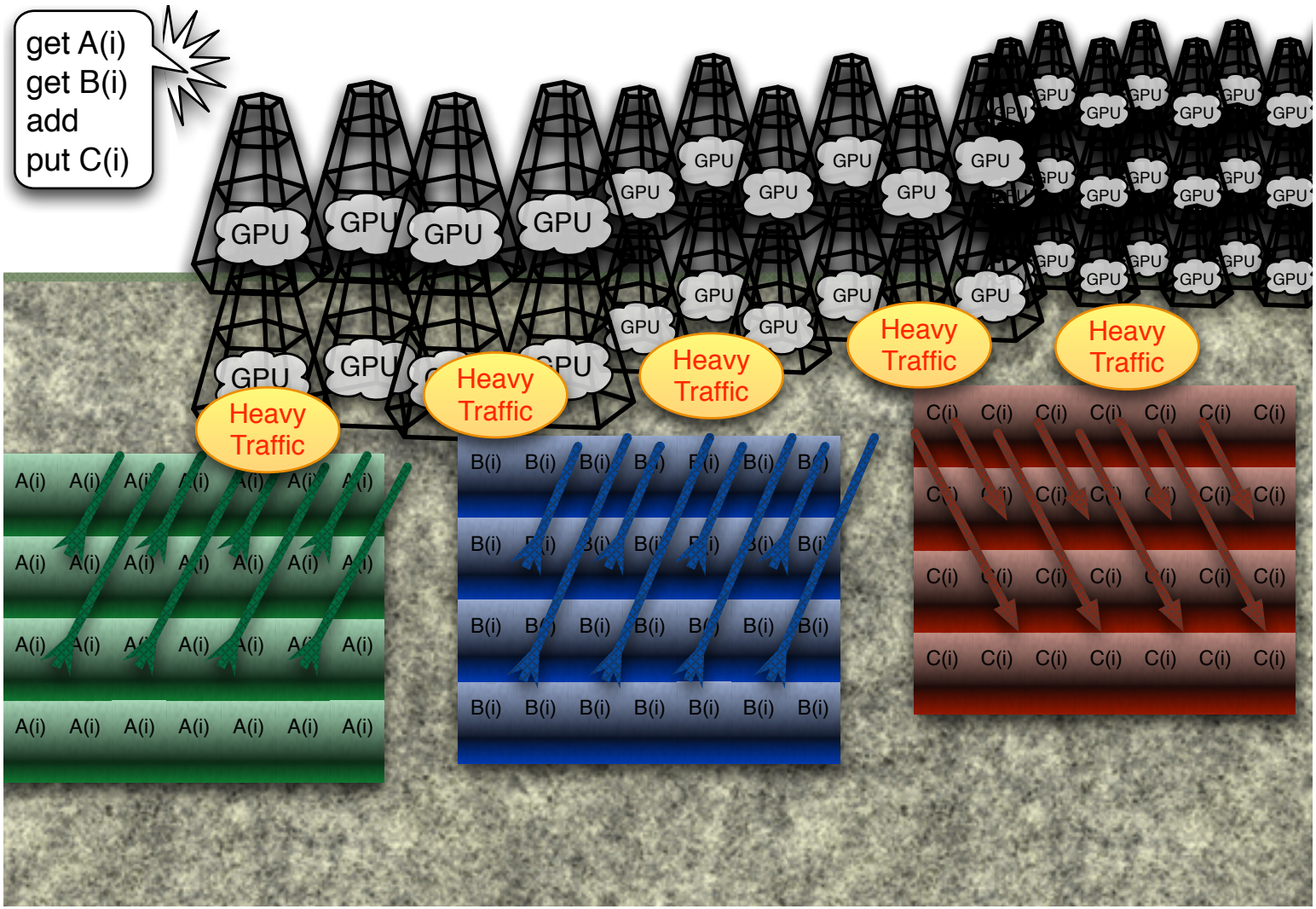
Partitioned data is naturally simple

- "SIMD" = Single Instruction Multiple Data
  - I.e., you want to do the same thing in each part of the data
- The "loop body" may have effects
- The effects will run in an arbitrary order
  - and this must not effect the outcome of the computation
- A full range of loop transforms can be applied
  - unroll/vectorize, buffer/prefetch, fork/join
- There may also be hardware efficiencies
  - One instruction unit broadcasting to several data pipes
  - (Not clear this is a big win, but I'm just a software guy.)

Split the data, keep one code stream



What could go wrong?

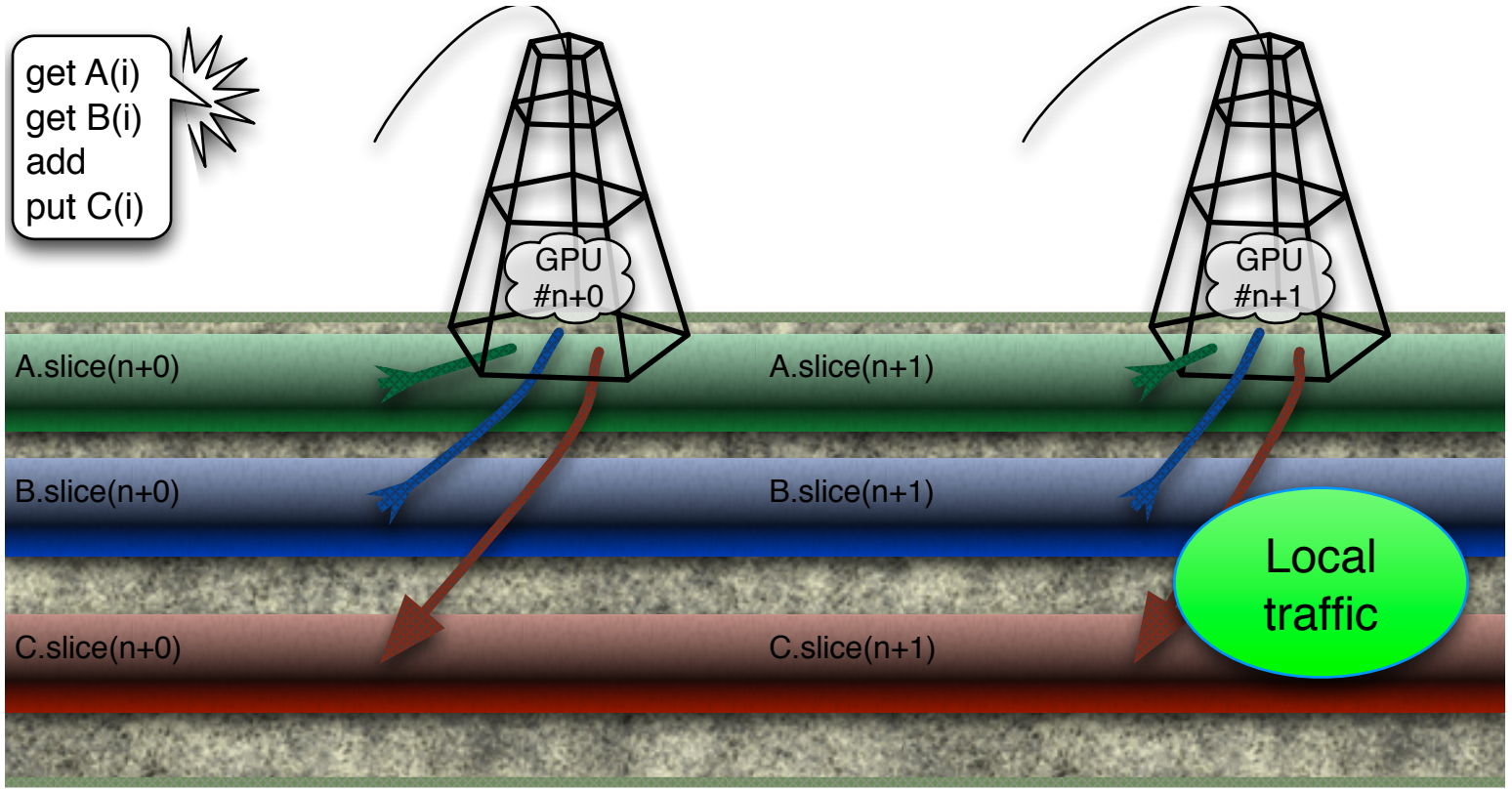


Communication dominates eventually

- Could express communication directly (pipes, MPI)
  - Does not leverage nice big shared memory – not JVM-ish
- Could add “placement” to types
  - Java “typefulness” is limited – `cpu2.new PlacedArray` ??
- Could add “placement” to objects
  - Would include explicit move (= “marshal”) operators
  - Feels like manual storage management; want to do better
- Contention: The JVM should control copying
  - GC “marshals” objects from newspace to oldspace
  - Moving data “near” to processing elements is JVM-ish job
  - JVM can use profiling, optimistic techniques, etc.

ORACLE

Let's stripe the data across the CPUs



Options for localizing data

- Manual (malloc/free): error prone
  - Also non-concurrent, which means chunk-sized latencies
  - Latencies make it hard to win compared with “JIT” fabrics
- Software caching/paging
  - JVM or some JDK framework manages local memories
- Hardware caching/paging
  - Yes, please. (Said the software guy.)
  - The option with the most low-level concurrency (prefetching)
  - This is a promising aspect of the HSA.

Options for localizing data (2)

- What if we don't have hardware caching?
  - Then we need rough equivalents
  - Need array-like objects which can be marshalled in chunks
  - Each chunk must be be separately bindable to a processor
  - Must also have a broadcast option (RTS instead of RTO)
- These features don't work with classic Java arrays!
- New arrays could have owner-thread safety rules
  - with runtime checking, same as for array bounds and null
  - would have to allow partitioning across multiple owners

ORACLE

What is a Java “GPU thread”?

- Must *not* be a subclass of `java.lang.Thread`!
  - Too much overhead from stack, etc.
- Minimally, it is a single iteration of the FORALL loop
  - Compare with other notions of “strand” or coroutine
  - Executes a (mainly linear) series of Java instructions
  - “Knows” which iteration it is doing (cf. GPU work item ID)
- Maximally, the whole loop, but cleverly decomposed
  - We don’t want to decompose Java “for” loops.
  - But Java (8) streams are designed to decompose!
- Somewhere in the middle, it is a “chunk” of iterations
  - Length is of the order |total size|/(#processors)
  - Library writers work with such loop chunks

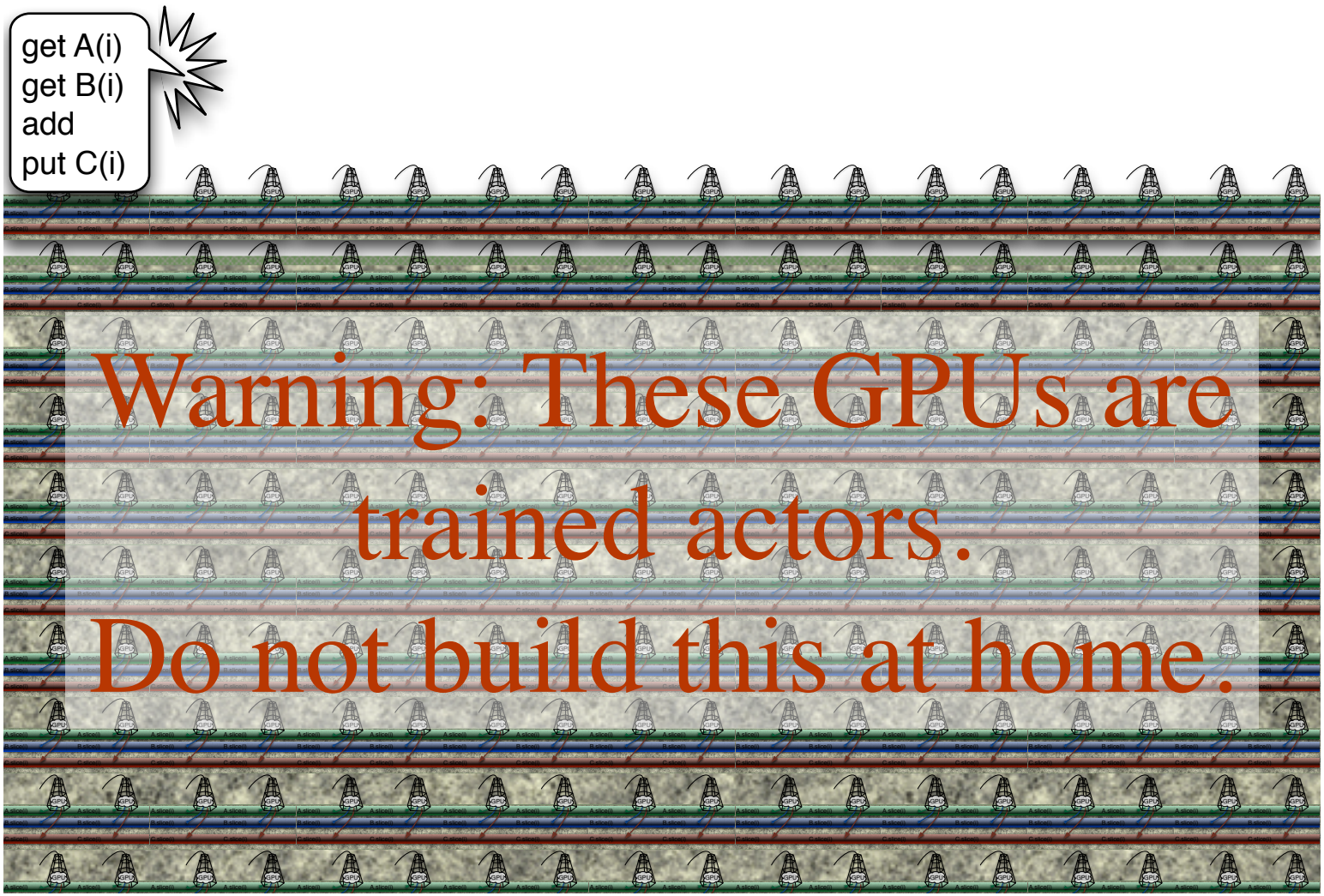
ORACLE

What is a Java “GPU thread”? (2)

- Virtual “GPU thread” should be as light as possible.
  - But, no user-visible semantic differences for bytecodes
- Suggestion: Do not reify, except as an index value
  - ...such as the result of `IntStream.range(0, N)`.
  - “Thread” execution consists of a lambda with parameter(s)
  - For 2D or 3D, could be a tuple of index values
- Moreover, `Thread.current` should return a dummy
  - Look to coroutine experiments for insight
- More detailed comparison with classic threads here:
  - http://wiki.openjdk.java.net/display/Sumatra/Thread+Swarms
- Perhaps this is a “thread swarm” or “warp”:
  - `s.parallel().forEach(x -> (#))`

ORACLE

This works — virtually



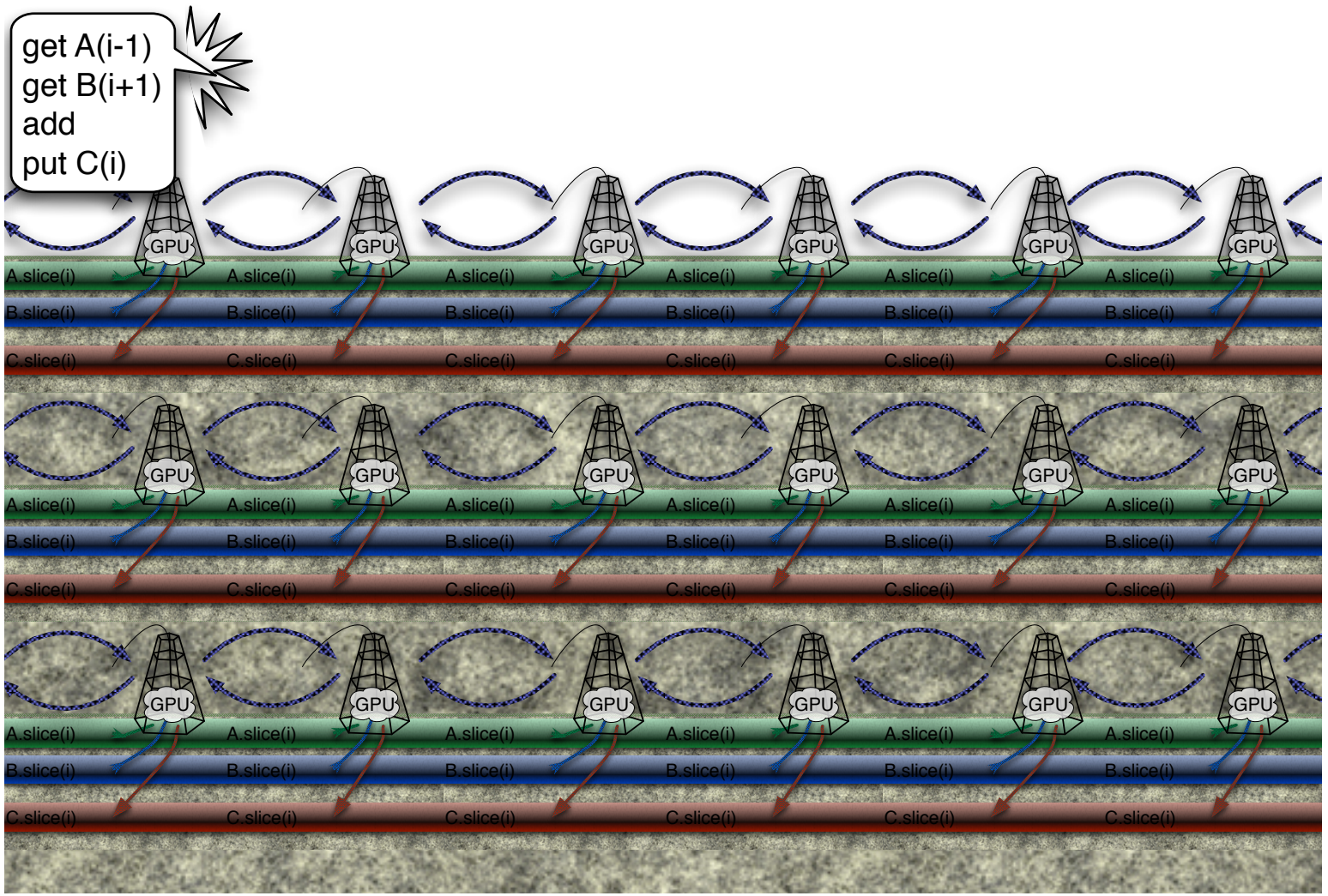
Issue: Placed data needs to be aligned

- In the running example, A, B, and C are aligned
  - If some chunk of A is partitioned onto some local memory...
  - ...then the corresponding chunk of B must follow
- Relates to the APL-ish idea of “shape conformance”
  - Also found in highly explicit languages like *Lisp and C*
  - Abstracted under the term “paralation” (Sabot)
- JDK 8 stream code generates a stream of indexes
  - The lambda (`i->A[i]+B[i]`) expresses A/B alignment
  - This is opaque to everybody but the JVM.
- Tentative corollary:
  Whatever “`Q[i]`” is, must be prefetchable/moveable

Regarding *local* communication

- The foregoing assumes completely “elemental” ops
  - C(i) depends only on A(i) and B(i)
- But the points apply just as well if neighbors can talk
  - A “stencil”: C(i) might depend on A(i-1), A(i+1), B(i-1), etc.
  - Loop transforms can deal with such mesh-like effects.
  - And many computational algorithms depend on meshes.

Mesh computing, with private memory



Slightly less local communication

- Non-neighbors can play too, if there are not too many
- Each iteration could do a heap access or table lookup
  - This works best if the looked-up data is sharable (RTS)
  - Sharability is one reason value objects are important
- Real GPUs need to simulate Java safety checks
  - NPE/RCE, div-by-zero or other errors will cause divergence
  - Rule of thumb? The less local the information, the more noisy

Global but log-scale communication

- The foregoing points also apply for reductions
  - Parallel reduction requires a O(log N) spanning tree.
  - Same for scans — JDK 8 `Arrays.parallelPrefix`
- JDK 8 reduction & scan works this way out of the box
  - including user-defined reduction ops!
- *Segmented* parallel prefix is a powerful building block
  - Can represented “nested” parallelism by flattening with keys
  - Might support the JDK 8 `groupingBy` collection operator

Regarding vectorization

- Modern hardware supplies efficient small vectors
  - small = cache-line granularity
  - efficient = SIMD across vector elements in a cycle or so
- These vectors operate independently of big arrays
  - but are a useful building block for all kinds of loops
  - some libraries build up mega-vectors from small ones
    - example: CVL (Blelloch)
- Open question: Does Java need a vector type?
  - Likely answer: Library writers may need them.
  - Likely answer: Compilers will want to emit them.
  - This is a use case for “value objects”.

Summary: Java liabilities / challenges

- Heavy threads
- Object state even when you don’t want it
  - Ditto for object locks and object reference identity
- Classic arrays (more wide-open state than Alaska)

Summary: Java assets

- Clean memory model
  - including final variables and eventually value types
  - model deeply supports concurrency
- O-O and/or ADT encapsulation
  - when we can wrap arrays we can tame them
- JDK 8 lambda-based streams
  - much good stuff right out of the box

Yes, but the question is...


ORACLE®