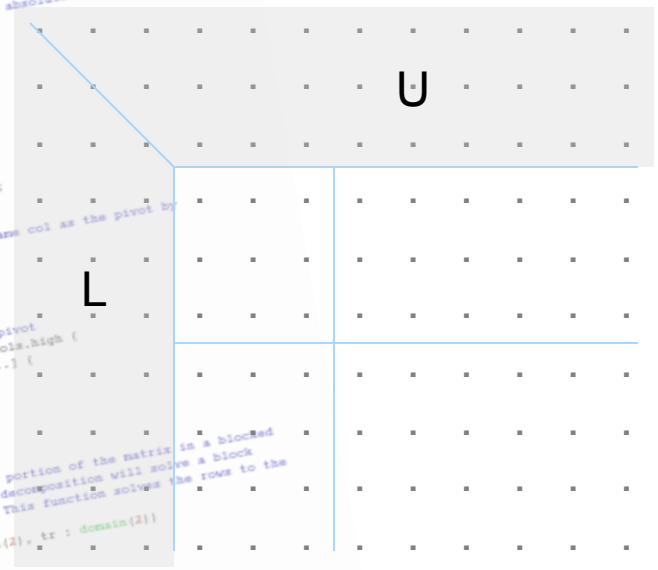


Andy Stone

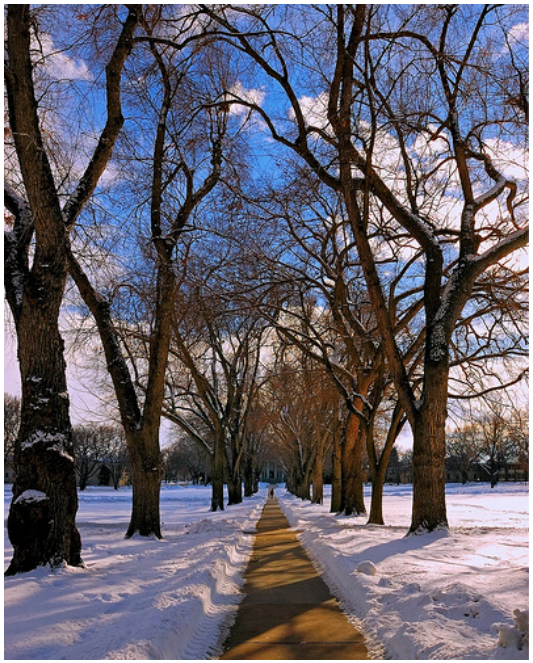
The HPL Benchmark in Chapel

Tiling Group

$$U_{i,j} = A_{i,j} - \sum_{k=1}^{i-1} L_{ik} U_{kj}$$

$$L_{i,j} = \frac{A_{i,j}}{U_{j,j}} - \sum_{k=1}^{j-1} L_{ik} U_{kj}$$

Background



Andy Stone

Grad student at Colorado State

Masters work is in
Program Analysis

Interests are:

- Compilers
- Languages
- Parallel Programming

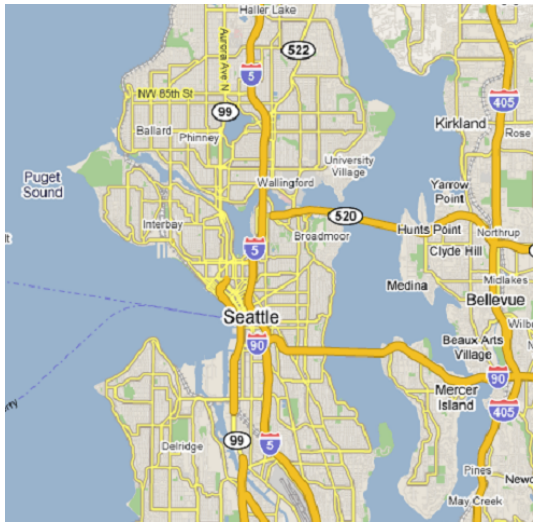
Openanalysis Toolkit

Photo from http://www.flickr.com/photos/fortphoto/349437386/

Spent Summer (12 weeks) in Seattle interning for Cray

Cray designs and manufactures Super Computers.

I worked with Brad Chamberlain, Steve Deitz, and Sam Figueroa in the Chapel group.



It didn't rain all that often, though it was overcast about half of the time.

About this talk

- My goal
  - Implement a working version of HPL in **Chapel**. **HPL** is a parallel, **blocked**, **LU decomposition** solver.
- This talk
  - Briefly talk about Chapel
  - Explain LU decomposition
  - Look at the blocking structure of **HPL**
  - Look at a distributed **matrix multiply** in Chapel

HPCCHALLENGE

Chapel's goals:

Increase programmer productivity

Produce efficient parallel code

HPCS goals:

Increase programmer productivity

Produce efficient parallel code

HPC Challenge, class II composition

4 benchmarks

3 written in Chapel when I showed up.

What remains:

1 **benchmark**, 1 **intern**, 1 **summer**

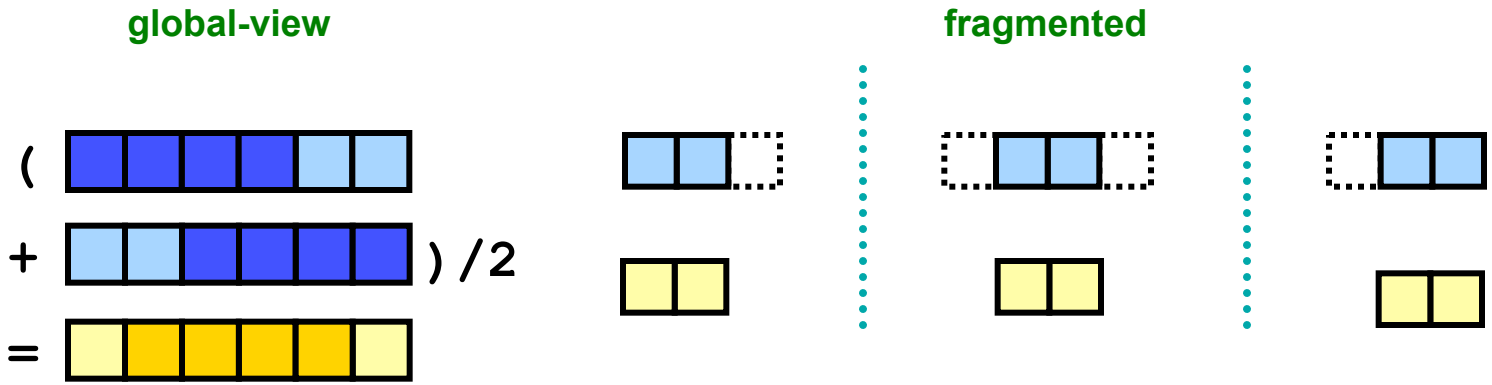
Chapel

Chapel

- **C**ascade **H**igh **P**roductivity **L**anguage
- A language and compiler for writing high-performance parallel applications
- Global view of computation
  - As opposed to SPMD style.
- Imperative
- Inspired from ZPL, HPF, Java, C#, MTA C/ Fortran extensions, and others

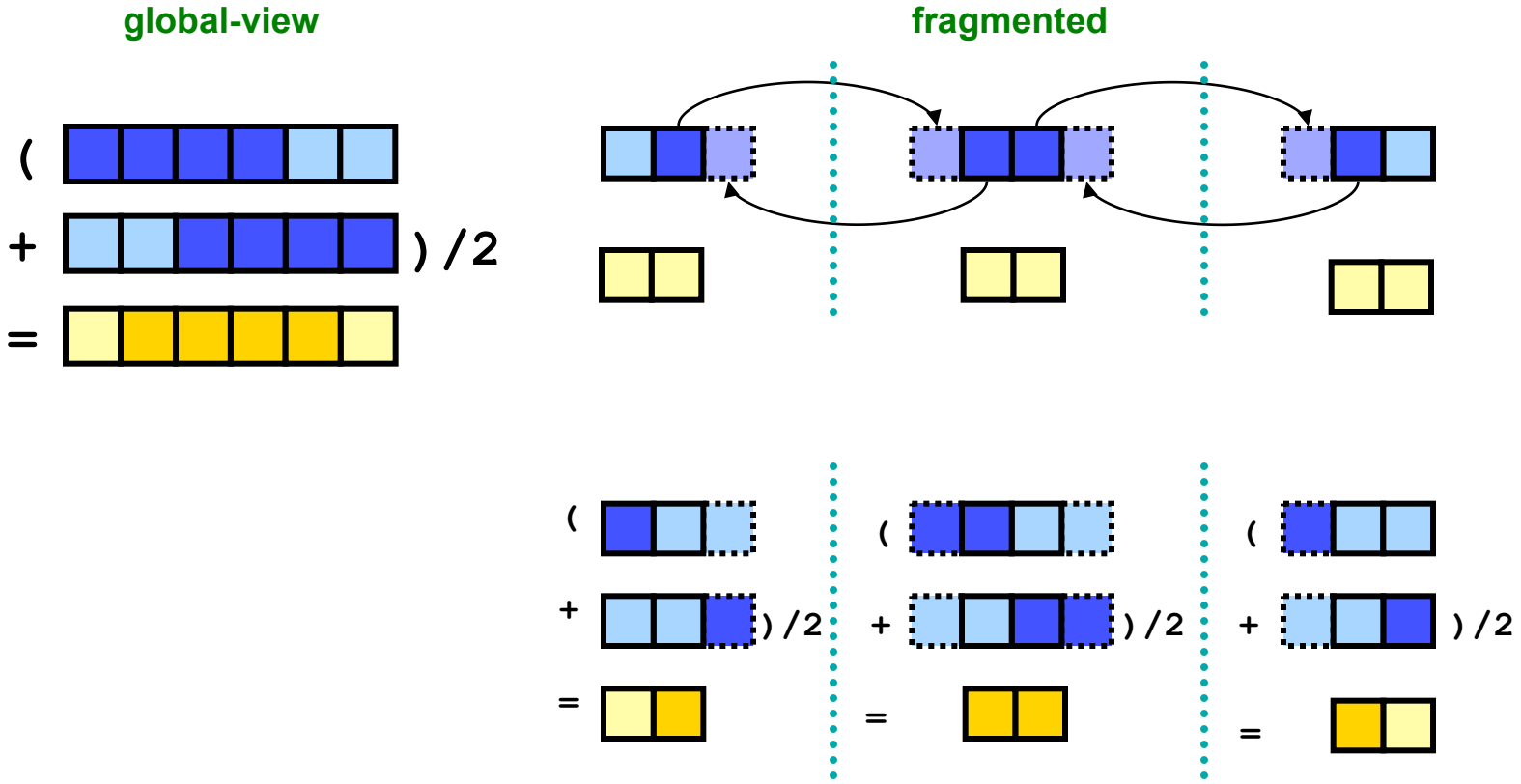
Global-view vs. Fragmented

Problem: “Apply 3-pt stencil to vector”



Global-view vs. Fragmented

Problem: "Apply 3-pt stencil to vector"



LU Decomposition

The Benchmark: Solving system of linear equations

Suppose you want to solve these equations:

$$5x + 3y - 2z = 23$$
$$7x + 9y + 3z = 102$$
$$8x + 8y - 8z = 8$$

You could put them into the form:

$$Ax = b$$

$$\begin{bmatrix} 5 & 3 & -2 \\ 7 & 9 & 3 \\ 8 & 8 & -8 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 23 \\ 102 \\ 8 \end{bmatrix}$$

Taking advantage of Properties of A

$$Ax = b$$

Properties of what A is can make this equation easier to solve.

If A is a **lower triangular** matrix we can solve using a simple method called **forward substitution**.

If A is an **upper triangular** matrix we can solve using a simple method called **back substitution**.

If we want to solve using back or forward substitution we need to massage the equation somehow to get that matrix to be upper or lower triangular.

LU Decomposition:

Let A be an $n \times n$ matrix.

Let L be a lower triangular matrix with a unit diagonal

Let U be an upper triangular matrix

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} = \begin{bmatrix} 1 & & & \\ l_{21} & 1 & & \\ l_{31} & l_{32} & 1 & \\ l_{41} & l_{42} & l_{43} & 1 \end{bmatrix} \cdot \begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ & u_{22} & u_{23} & u_{24} \\ & & u_{33} & u_{34} \\ & & & u_{44} \end{bmatrix}$$

Using LU

So we use LU decomposition, then:

$$Ax = b$$

Turns into:

$$LUx = b$$

Using LU

$$LUx = b$$

Is nice and all but still not in the form we need. What we really need is something like:

$$Lx = b \quad \text{or} \quad Ux = b$$

Using LU

What if we say:

$$\text{Let } y = Ux$$

Then $LUx = b$ Could be written as: $Ly = b$

Which we can solve for with **forward substitution**, if we know what y is.

Using LU

And since we said:

$$\text{Let } y = Ux$$

We can solve for y using **backward substitution!**

Coming up with an LU algorithm

Our strategy: Look at the equations

A=LU is a matrix multiply, which can be written as

$$A_{i,j} = \sum_{k=1}^{\min(i,j)} L_{ik} U_{kj}$$

Coming up with an LU algorithm

A=LU, can also be expressed as:

$$A_{i,j} = \begin{cases} i \le j: & U_{ij} + \sum_{k=1}^{i-1} L_{ik} U_{kj} \\ j < i: & L_{ij} U_{jj} + \sum_{k=1}^{j-1} L_{ik} U_{kj} \end{cases}$$

Coming up with an LU algorithm

Solving for L and U:

$$\text{if we assume } i \leq j:$$

$$U_{i,j} = A_{ij} - \sum_{k=1}^{i-1} L_{ik} U_{kj}$$

$$\text{if we assume } j < i:$$

$$L_{i,j} = \frac{1}{U_{j,j}} \left(A_{i,j} - \sum_{k=1}^{j-1} L_{ik} U_{kj} \right)$$

A simple LU-decomp algorithm:

```
def luDecomp(n : int, A : [1..n,1..n] real) {
    for k in 1..n {
        for i in k+1..n {
            A[i,k] /= A[k,k];
            for j in k+1..n {
                A[i,j] -= A[i,k] * A[k,j];
            }
        }
    }
}
```

Transforms A into new matrix that combines L and U

A Blocked Algorithm

The Block LU Algorithm

Blocking is a technique used in many linear-algebra computations to make them more efficient.

You'll see why its called blocking as we go into more detail on the algorithm.

While the matrix is not factored {

Grab a block in top-left portion of unfactored matrix.

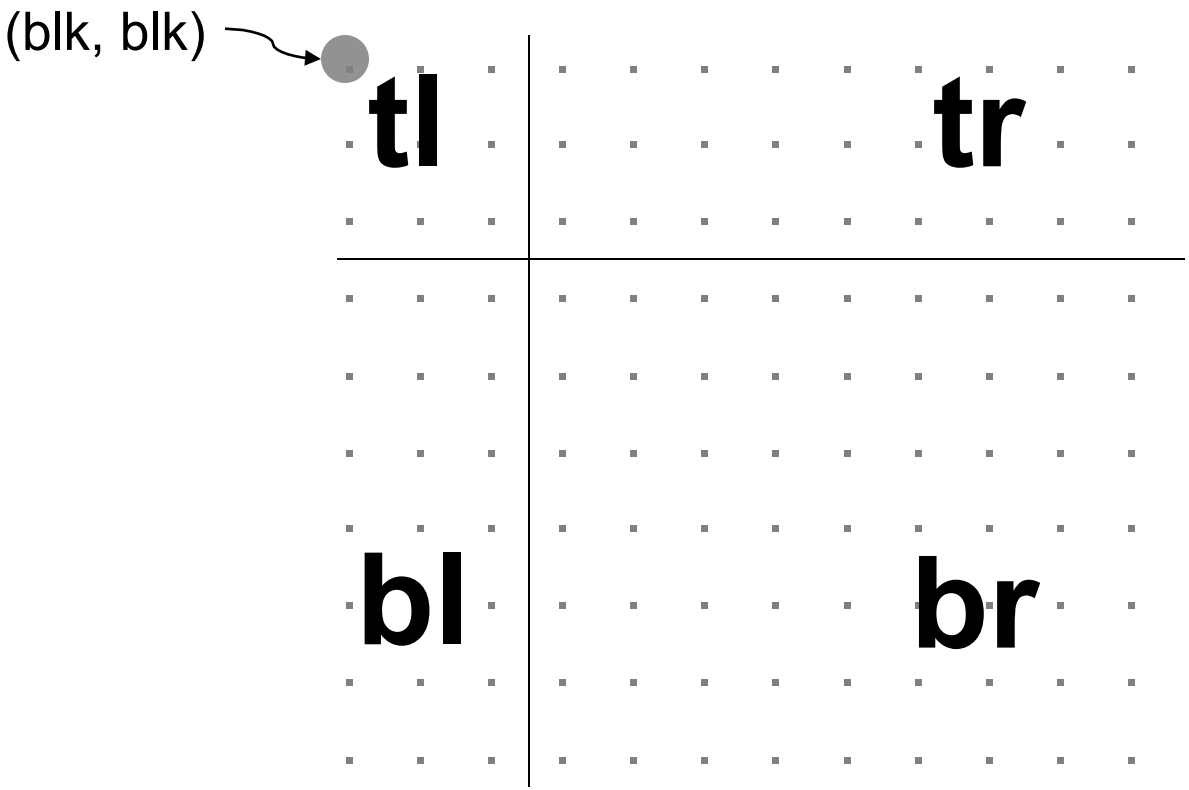
Panel solve the block and the elements below it.

Row update the elements to the right

Calculate **Schur Complement** to update the trailing elements.

}

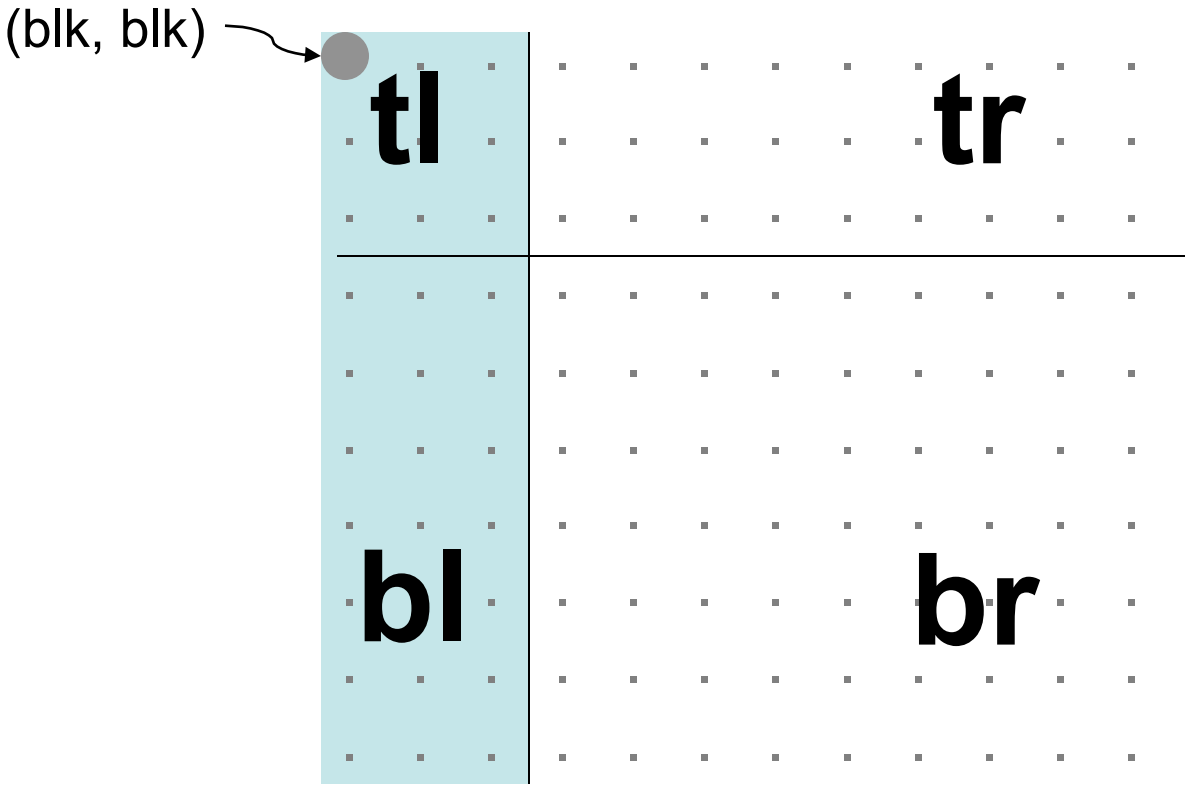
The Algorithm: Animated



Grab a block in top-left portion of unfactored matrix.

The block is the region marked tl

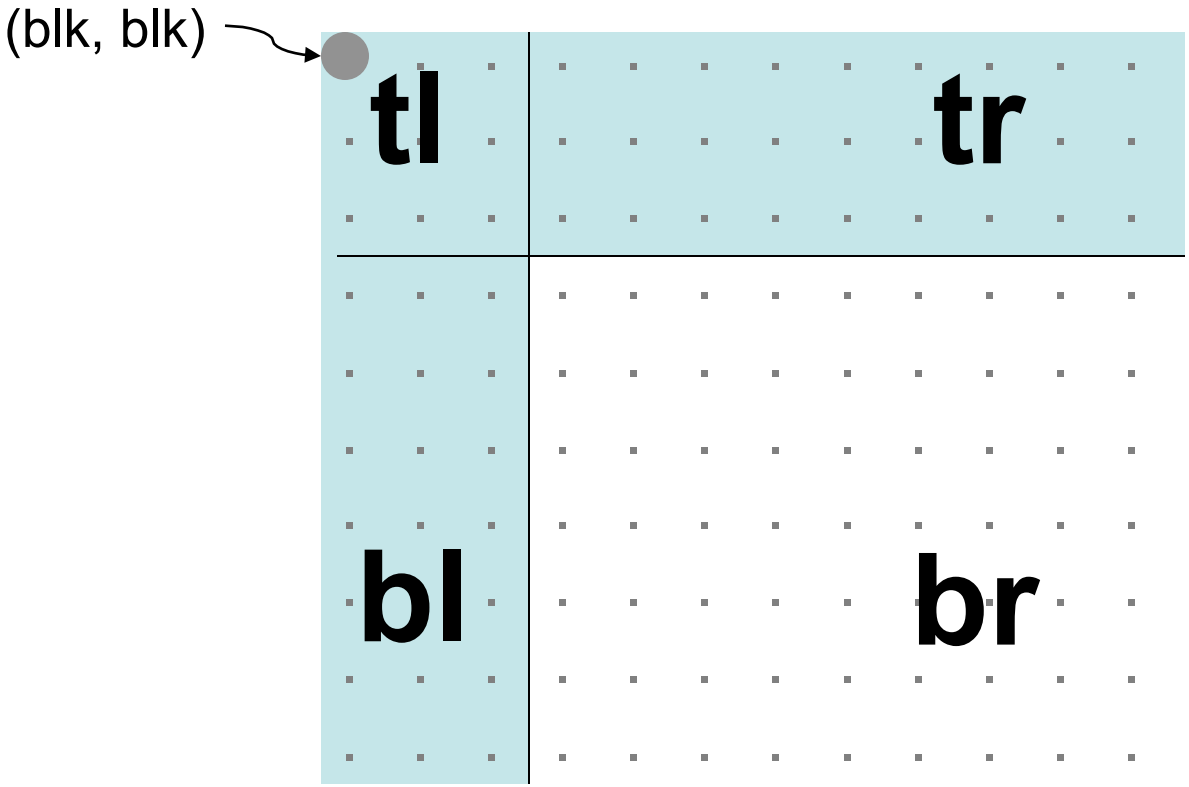
The Algorithm: Animated



Panel solve the block and the elements below it

The elements below are those in the region bl

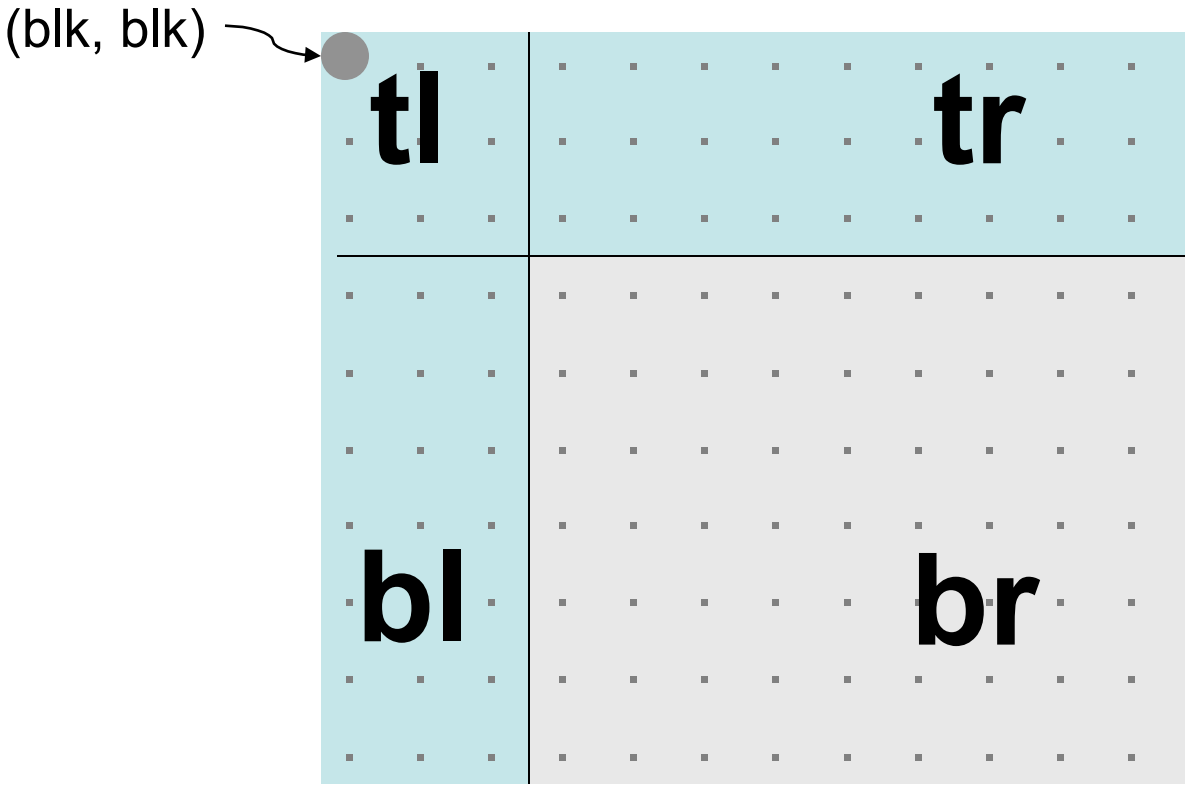
The Algorithm: Animated



Row update the elements to the right

Which would be those in the region tr

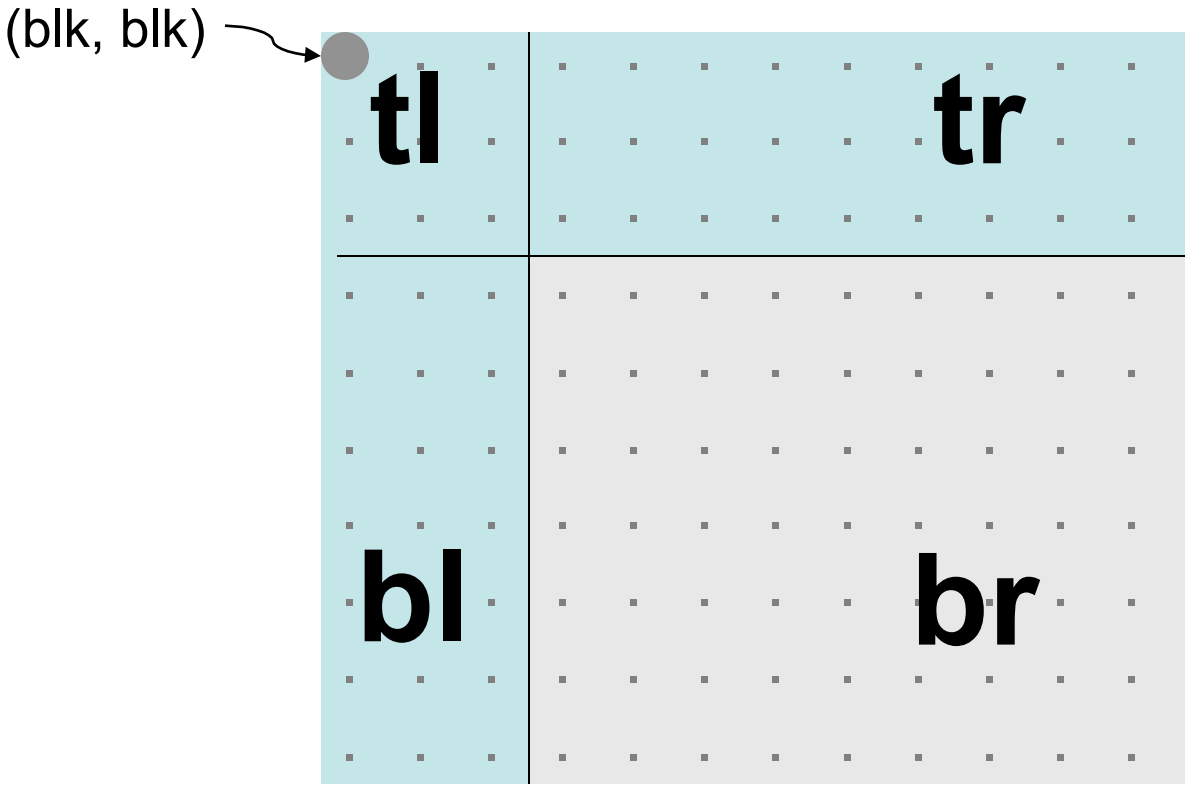
The Algorithm: Animated



Do a matrix multiply to update the trailing elements

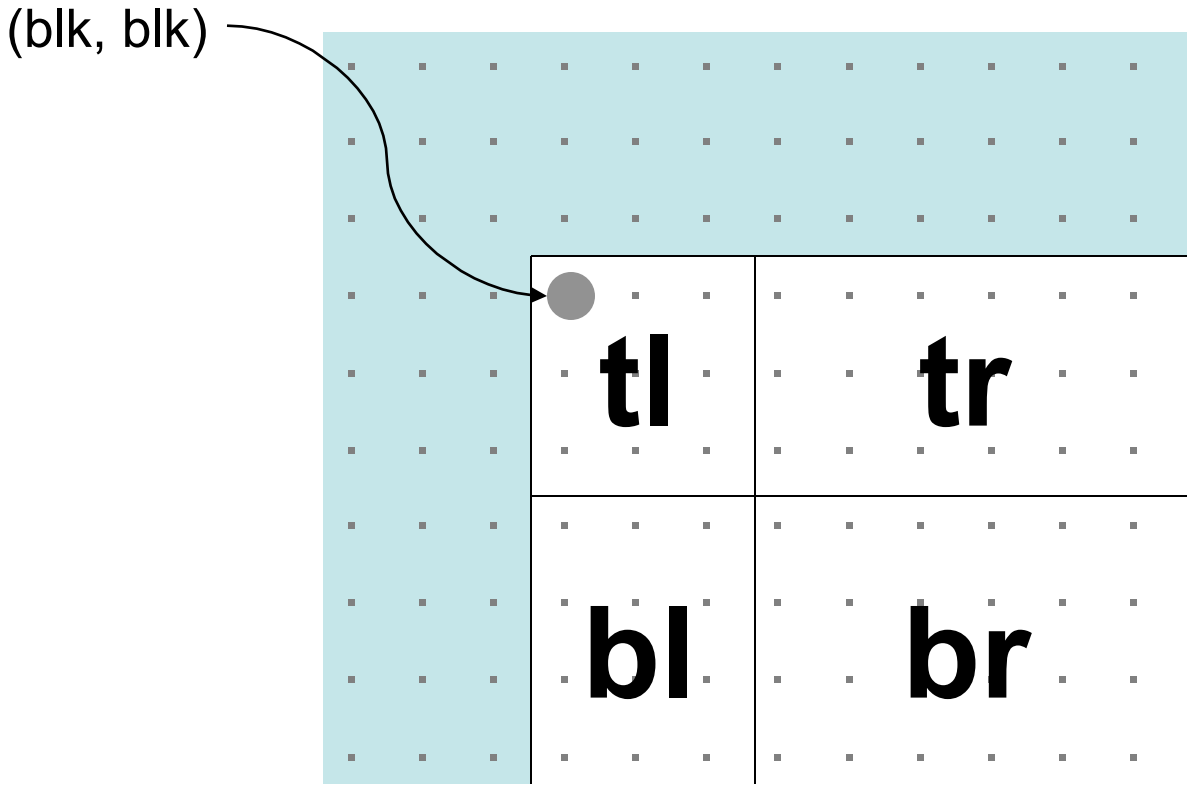
The elements in **bl** are multiplied by the elements **tr** to form **br**

The Algorithm: Animated



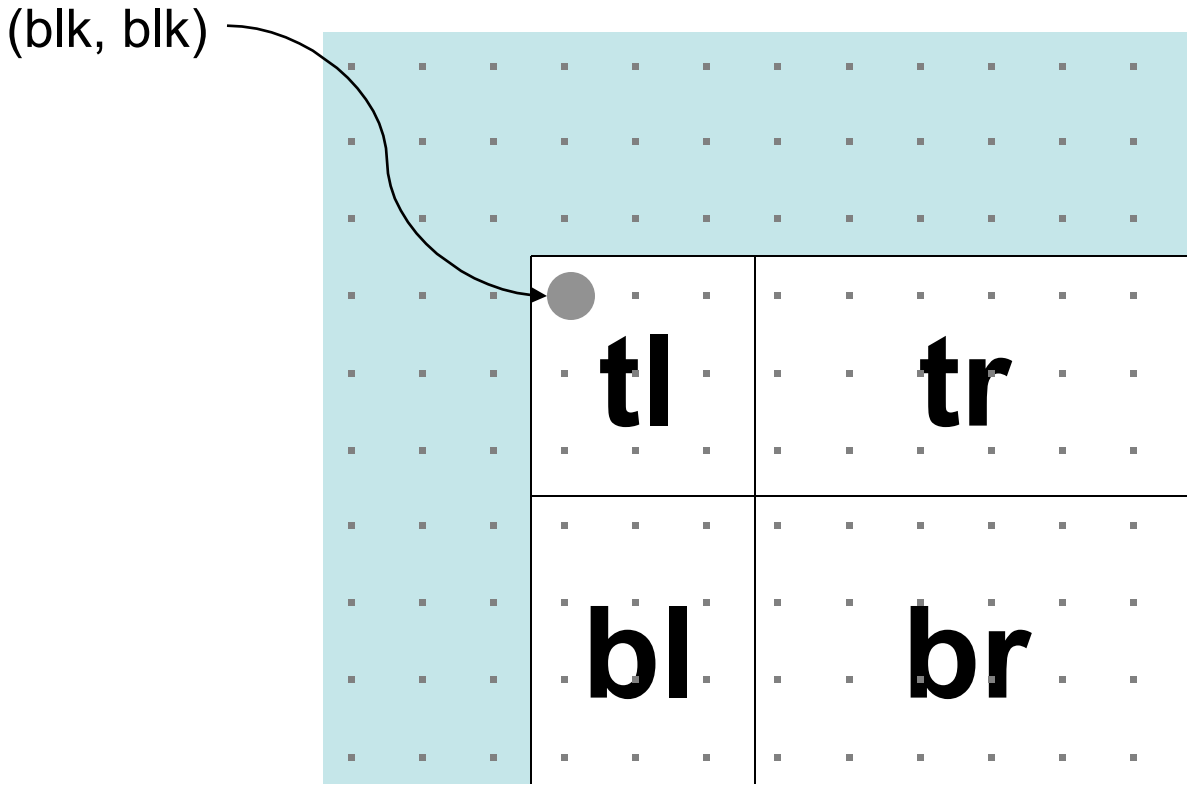
At this point regions **tl**, **tr**, and **bl** are factored and region **br** is ready to be updated by applying the algorithm recursively.

The Algorithm: Animated



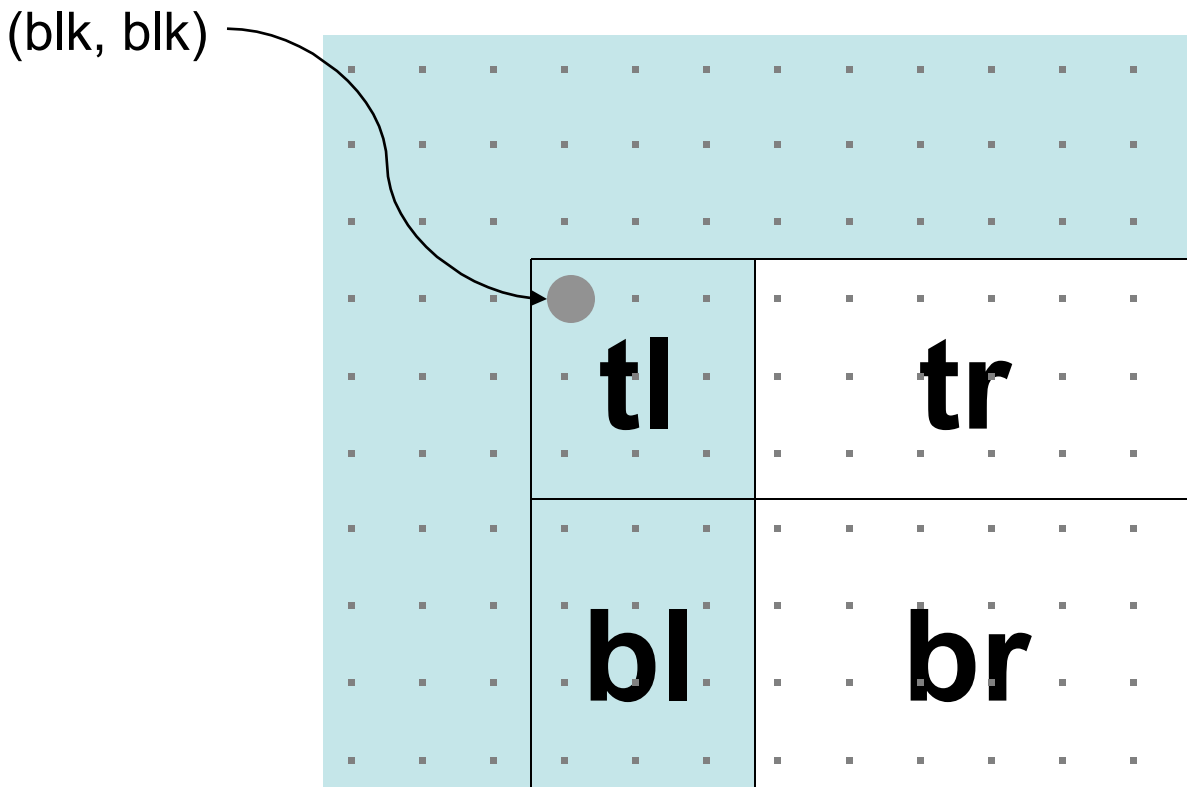
So we take what was the former br region and partiton that!

The Algorithm: Animated



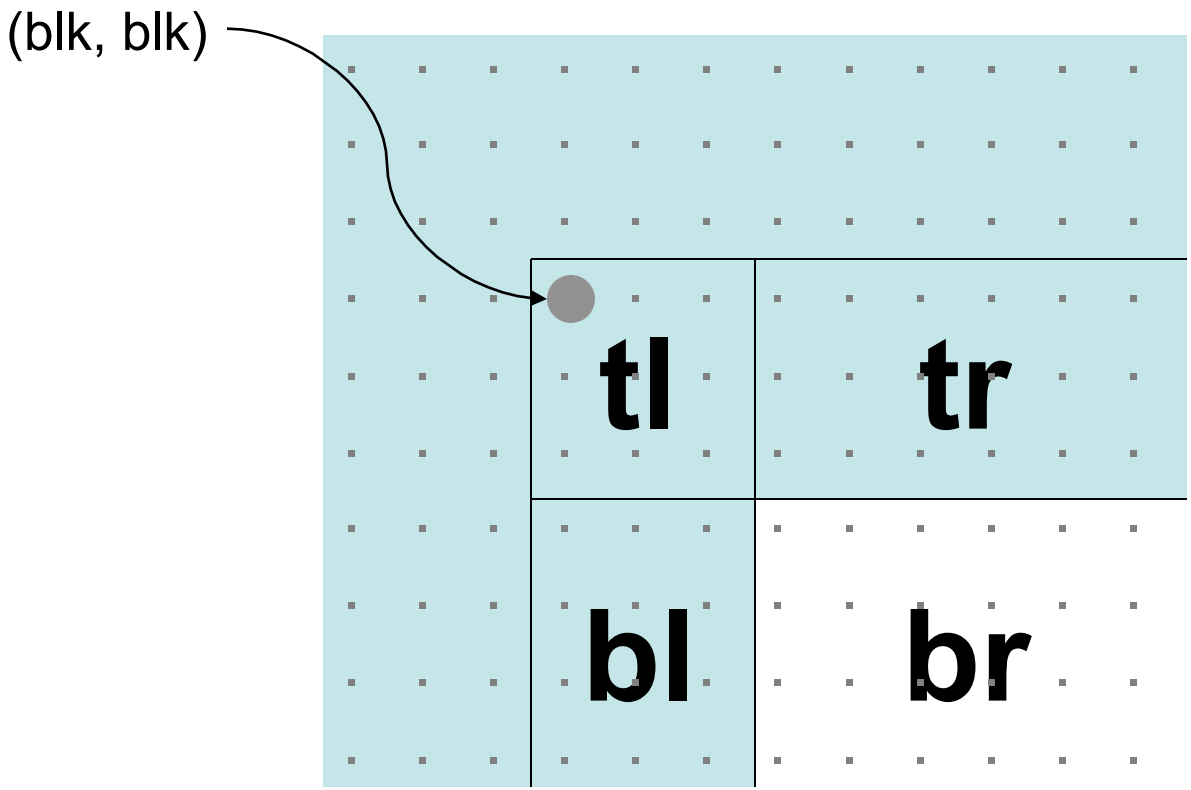
And the process repeats

The Algorithm: Animated



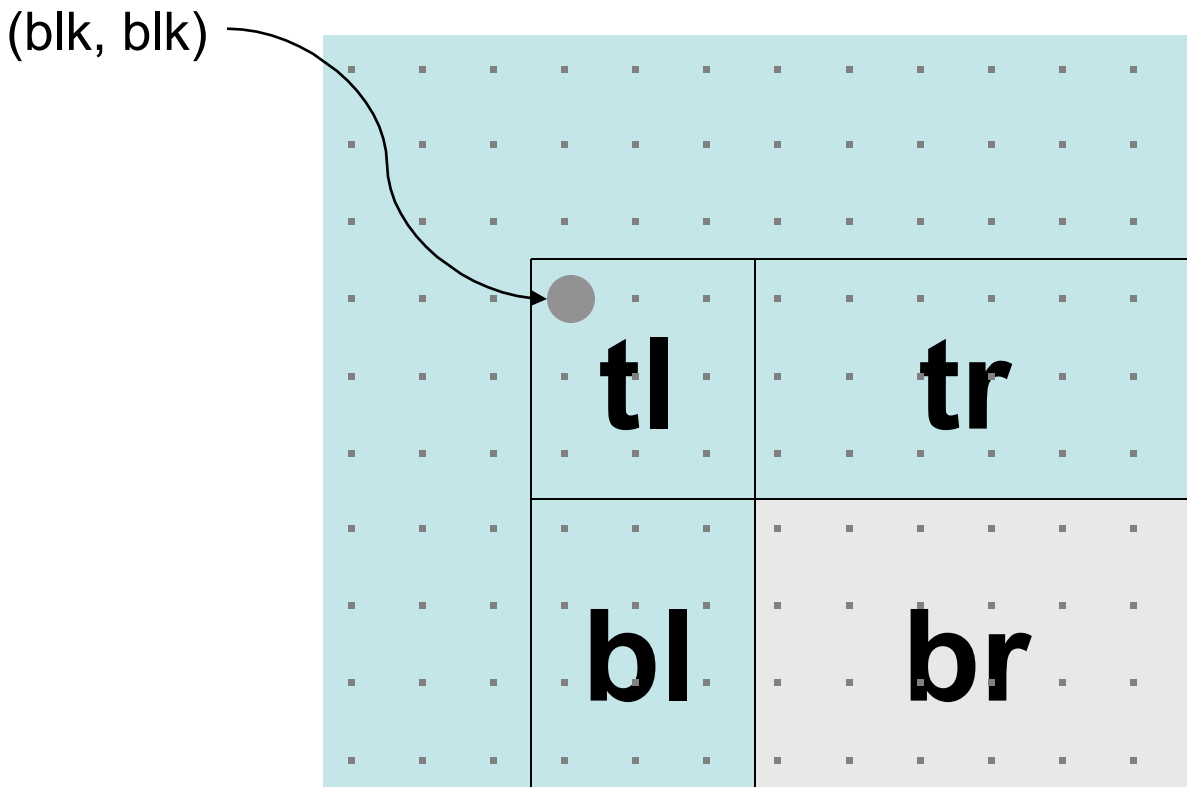
And the process repeats

The Algorithm: Animated



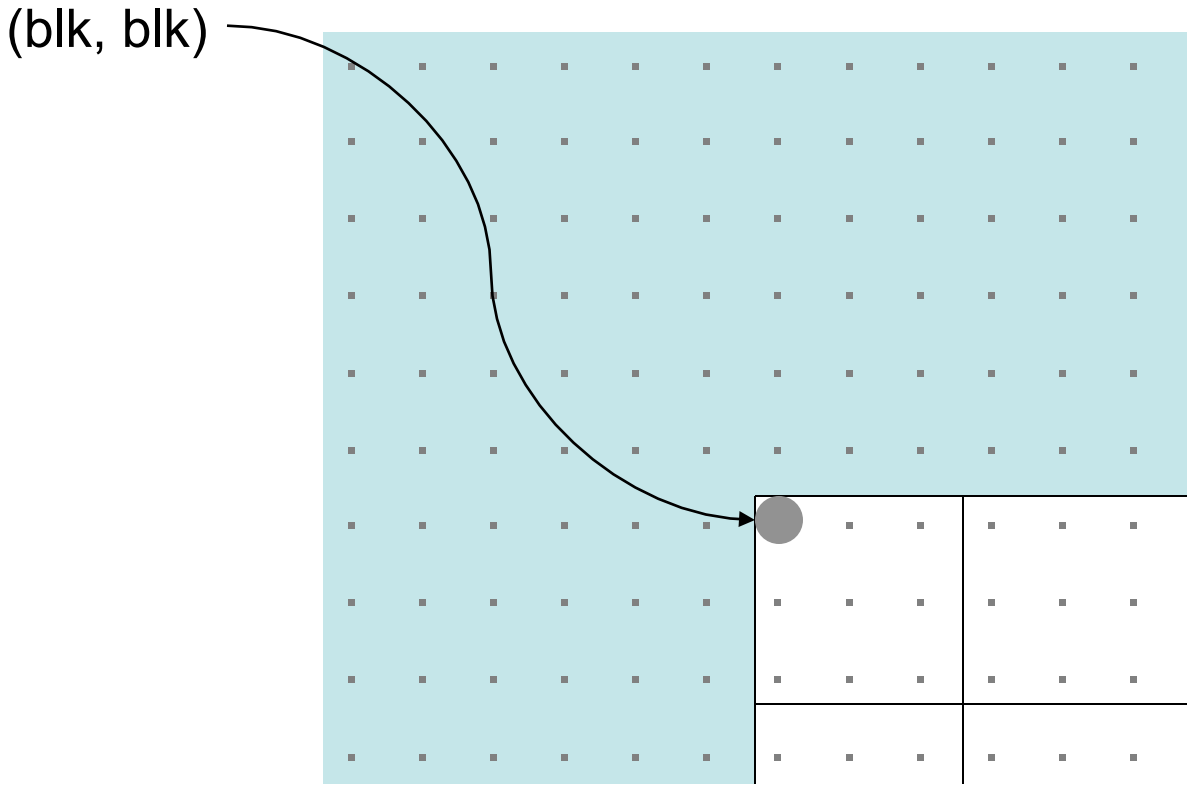
And the process repeats

The Algorithm: Animated



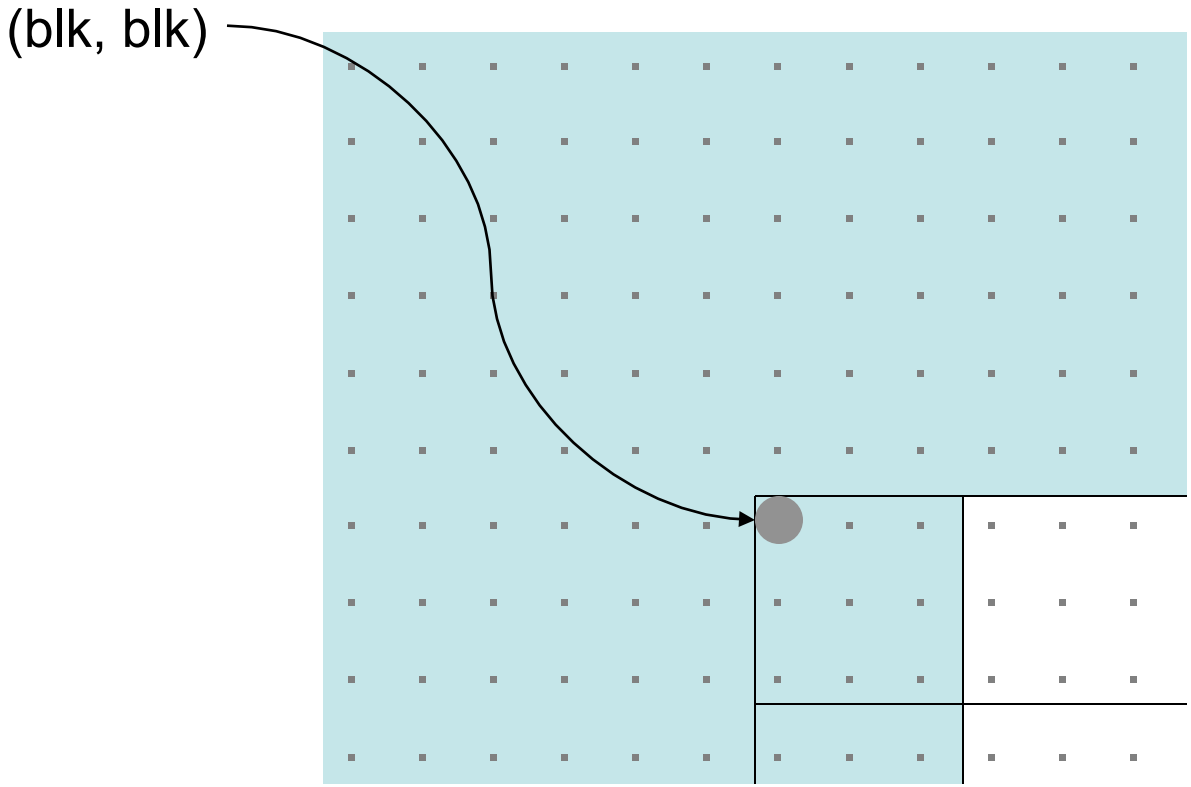
And the process repeats

The Algorithm: Animated



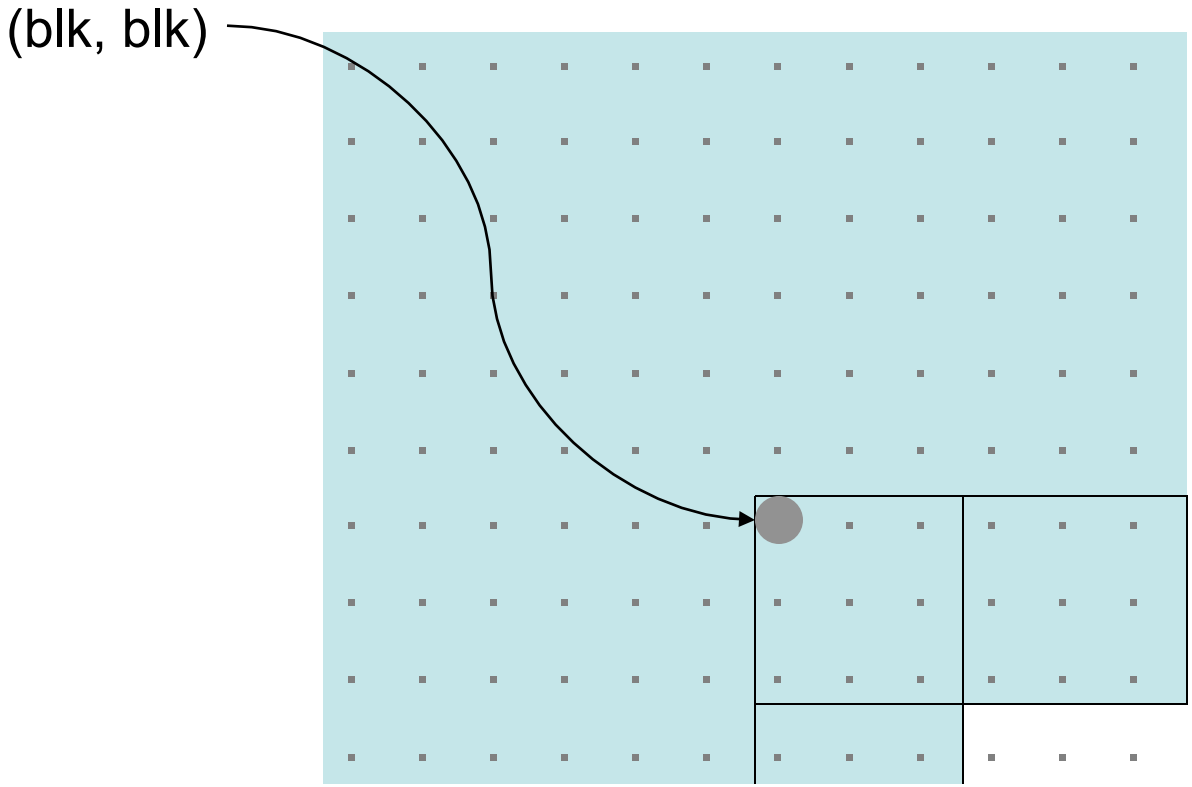
And it repeats again!

The Algorithm: Animated



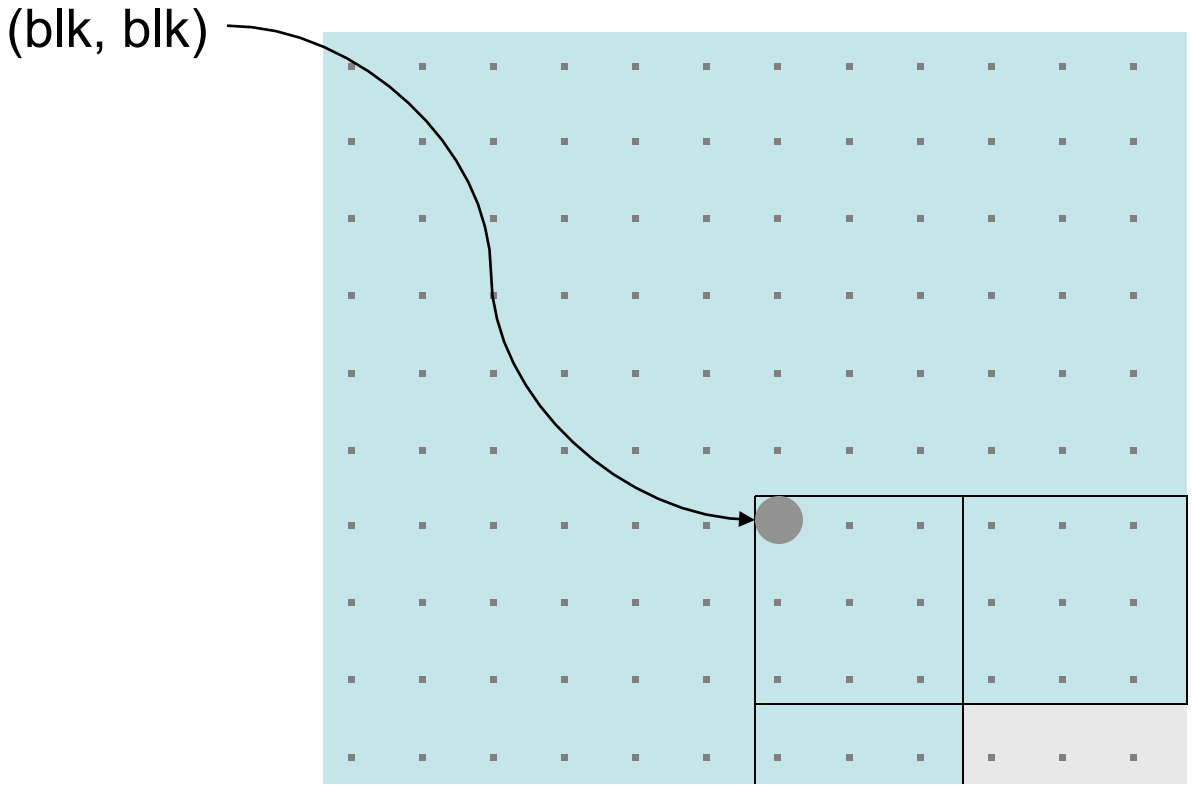
And it repeats again!

The Algorithm: Animated



And it repeats again!

The Algorithm: Animated



And it repeats again!

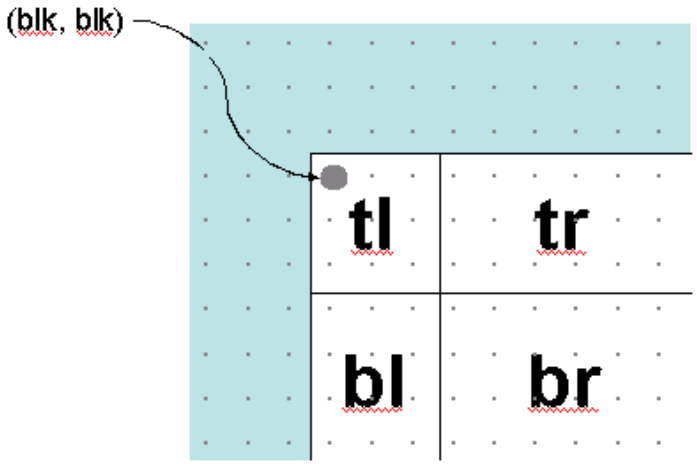
Chapel Domains

```
var AD : domain(2) = [1..n, 1..n];
var A : [AD] real(64);
```

Subdomains

```
var SubD : subdomain(AD) =
    [1..n/2, 1..n/2];
```

Code to partition




```
for blk in 1..n by blkSize {
    var tl = AD[blk..#blkSize, blk..#blkSize];
    var tr = AD[blk..#blkSize, blk+blkSize..];
    var bl = AD[blk+blkSize.., blk..#blkSize];
    var br = AD[blk+blkSize.., blk+blkSize..];
    var l  = AD[blk.., blk..#blkSize];

    // ...

}
```

Blocking Algorithms: Motivation

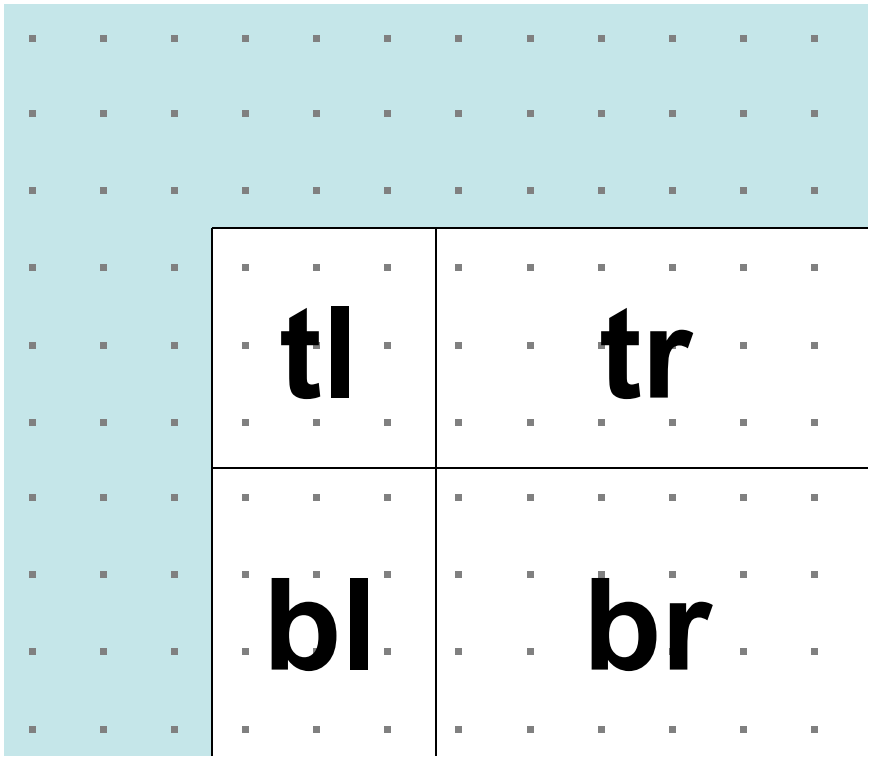
For HPL:
Computation time: O(n^3)
Memory usage: O(n^2)

Each computation accesses (reads or writes to memory) at-least once, so its pretty clear that the same location of memory must be accessed more than once (Pigeon-hole principle).

The Schur Complement

A distributed matrix multiply in Chapel

The Matrix Multiply



We need to multiply region **bl** by region **tr** to update region **br**

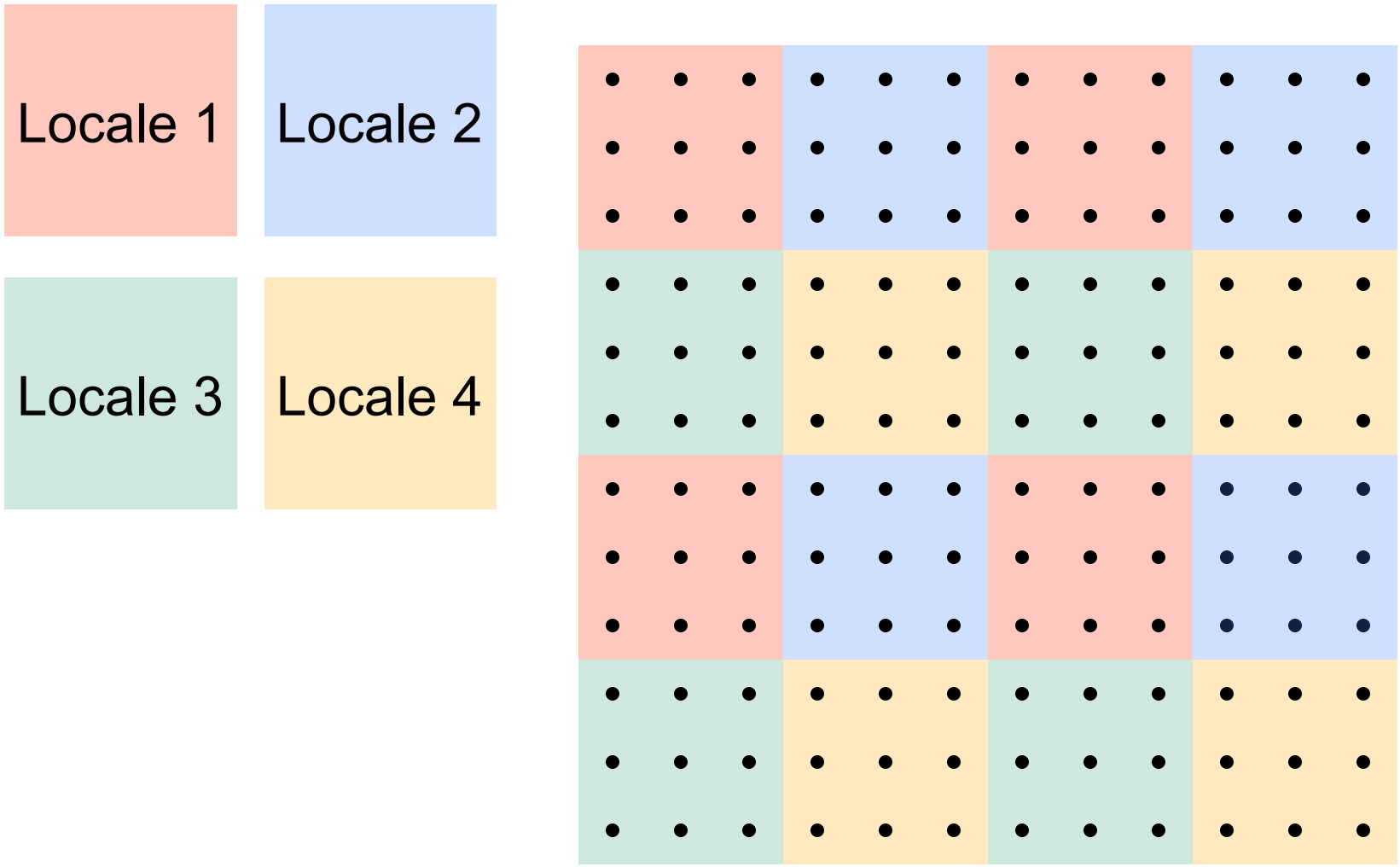
Most computation time spent in this part of the algorithm

This is what we should really try and make efficient.

Keep in mind the matrix is distributed

There are many ways to do this part, I'll show you two.

A matrix distributed in blocks

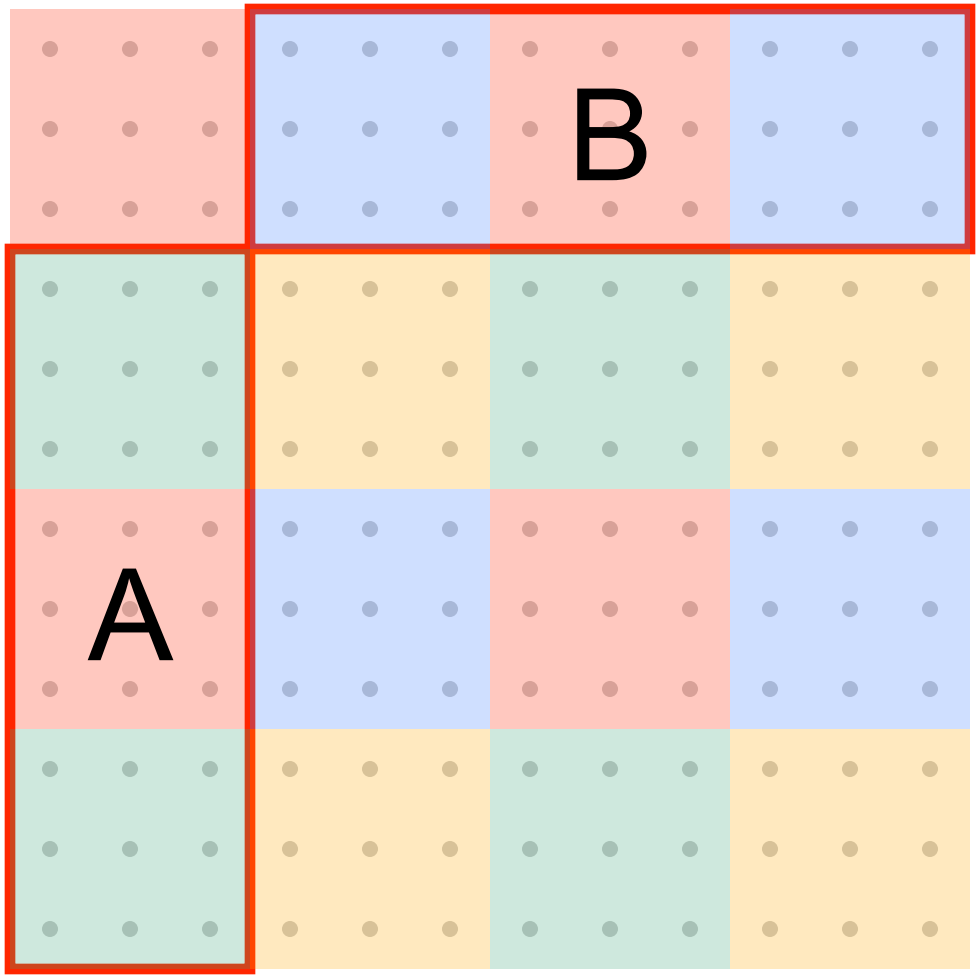


Unimplemented

Applying a distribution to an array

```
var D1 : domain(2) distributed
     BlkCyclic2D(blkSize)= [1..n, 1..n];

var D2 : domain(2) distributed
     Dimensional(
          BlkCyc(blkSize), BlkCyc(blkSize));
D2 = [1..n, 1..n];

var A : [D2] real;
```

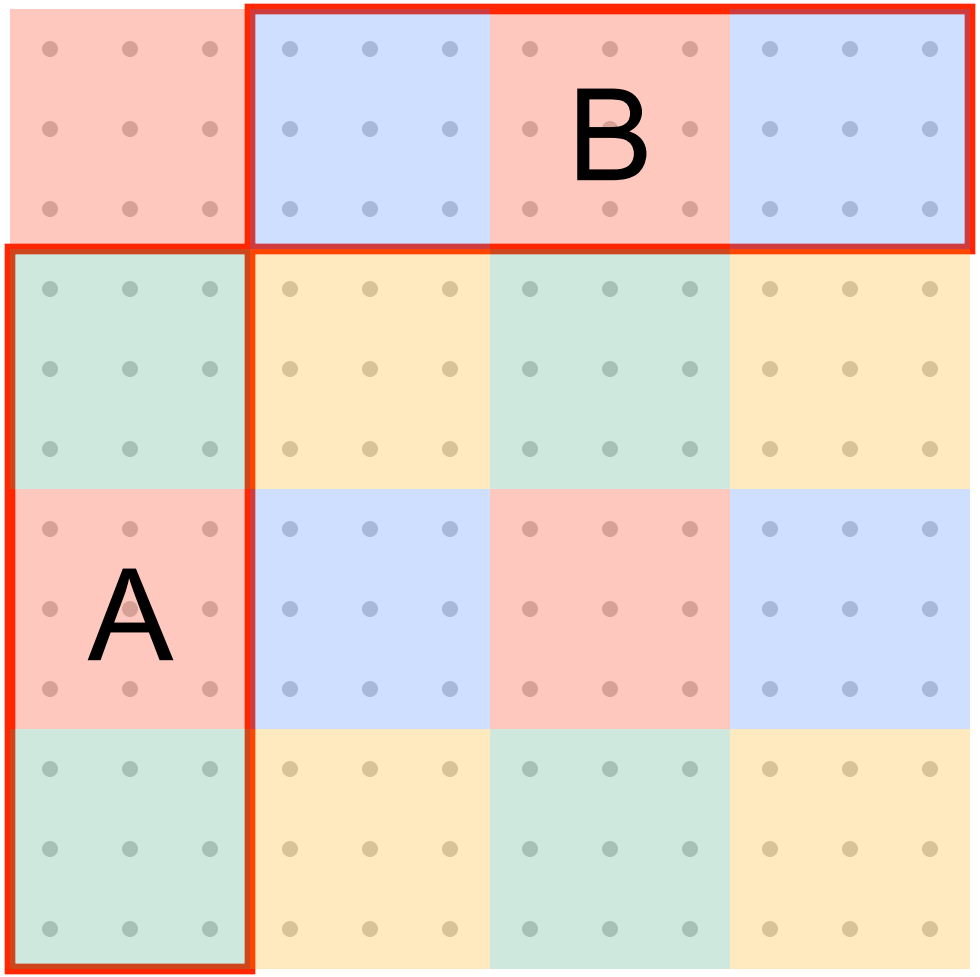
Some great properties about MM for HPL



The algorithm's blocks fall right along the blocks that distribute the matrix!

We need to multiply a column of blocks (**A**) by a row of blocks (**B**) to update a matrix of blocks.

Strategy for matrix-mult

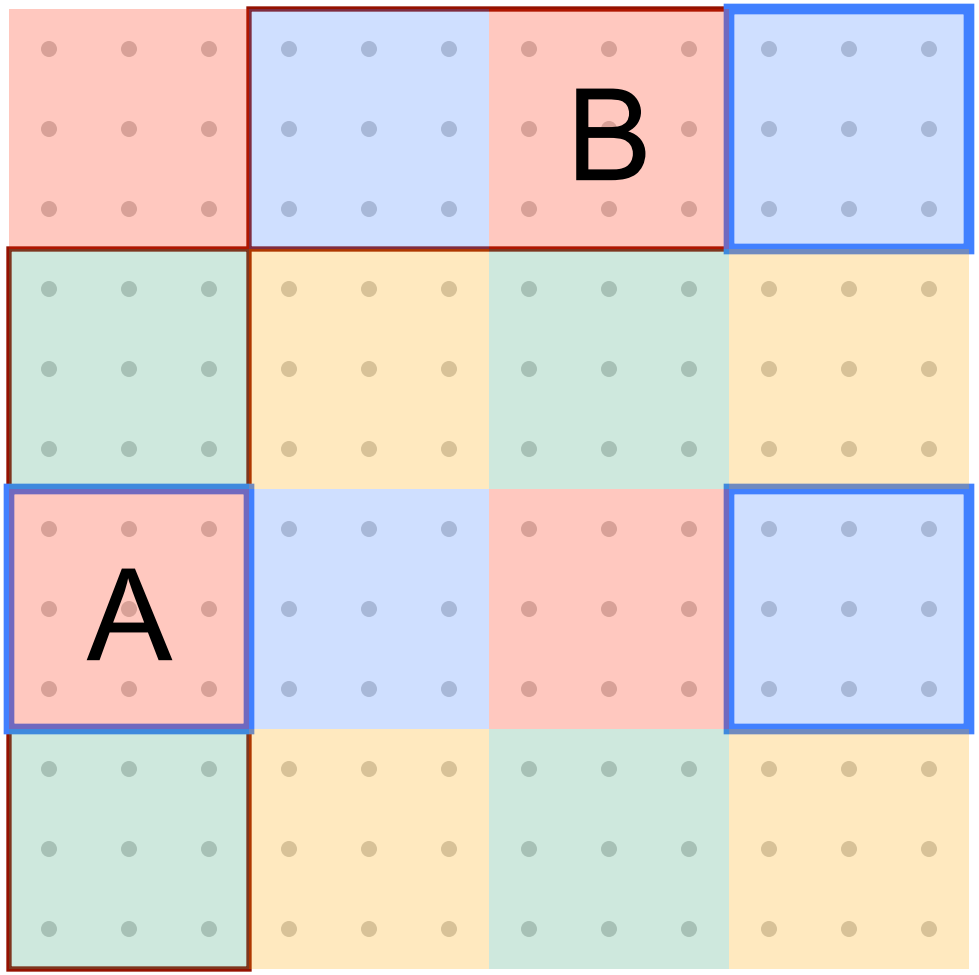


A Simple Strategy:

Replicate data so each locale has a local copy of the information it needs.

Perform matrix multiply on every locale concurrently.

Strategy for matrix-mult



Each locale will need the data from the block in tl in the same row and the block in tr in the same column.

The matrix-mult algorithm

Three step process:

1. Broadcast the data in A to the locales to its right
2. Broadcast the data in B to the locales below it
3. Have all locales perform a local matrix-multiply with the data it now has

The matrix-mult algorithm

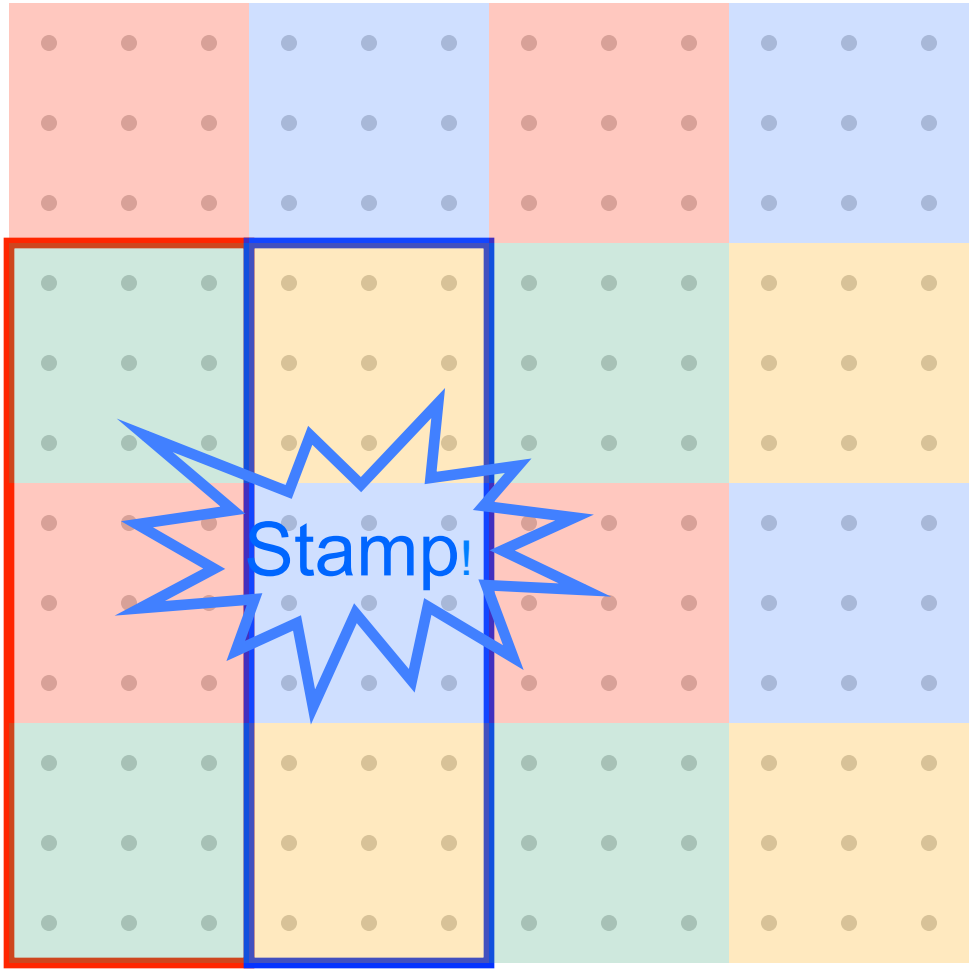
Three step process:

1. Broadcast the data in A to the locales to its right
2. Broadcast the data in B to the locales below it

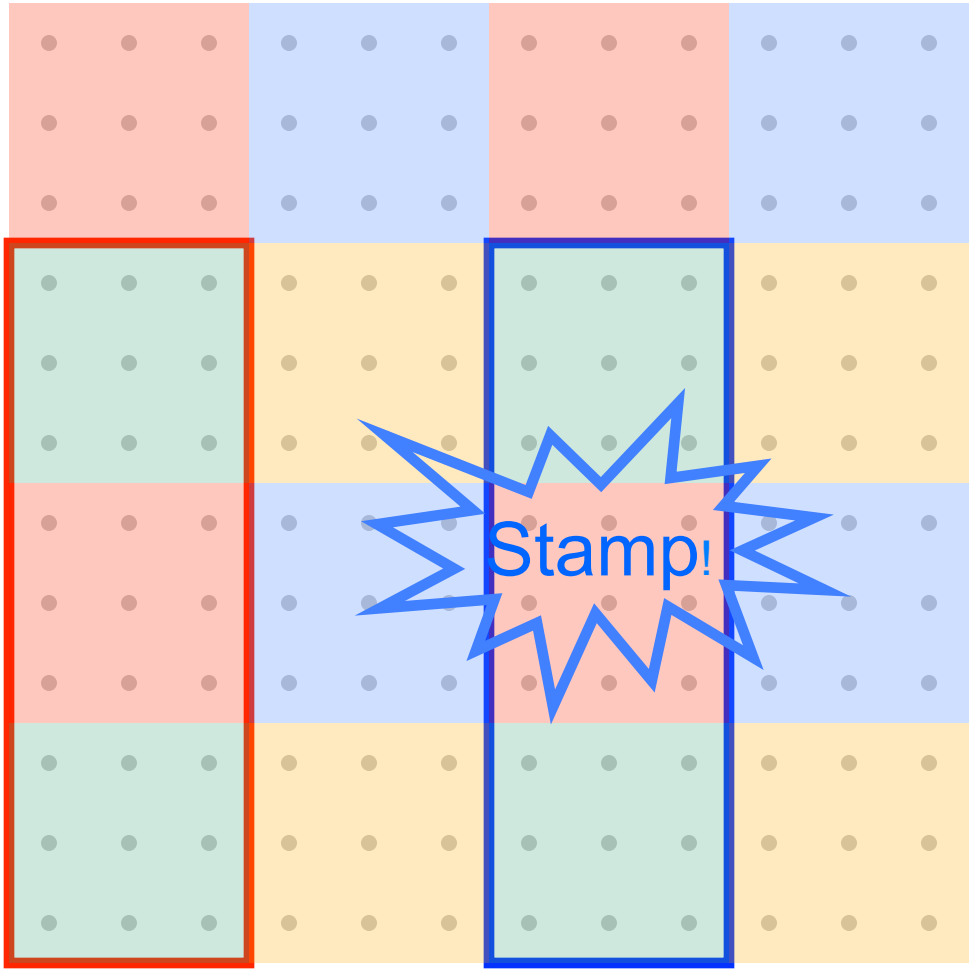
Stamping steps

3. Have all locales perform a local matrix-multiply with the data it now has

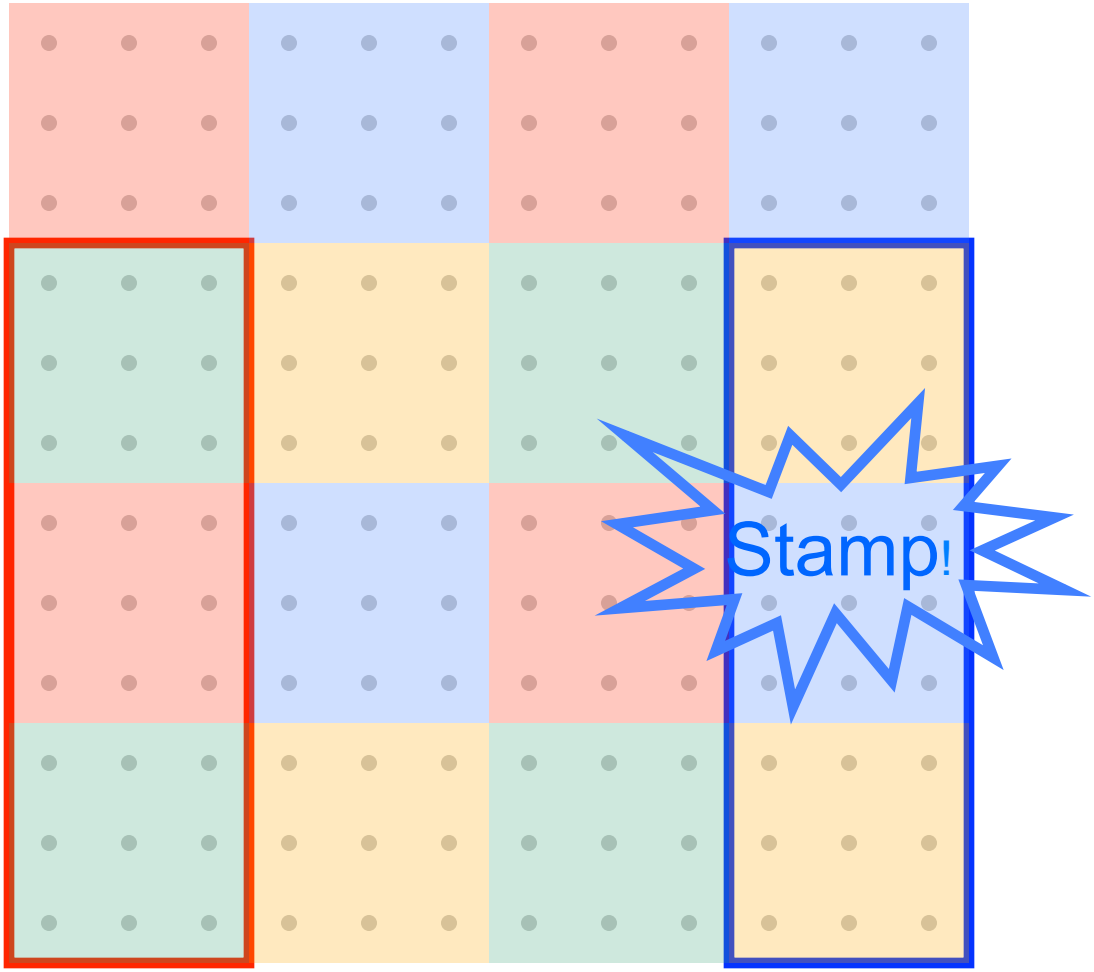
Stamping A



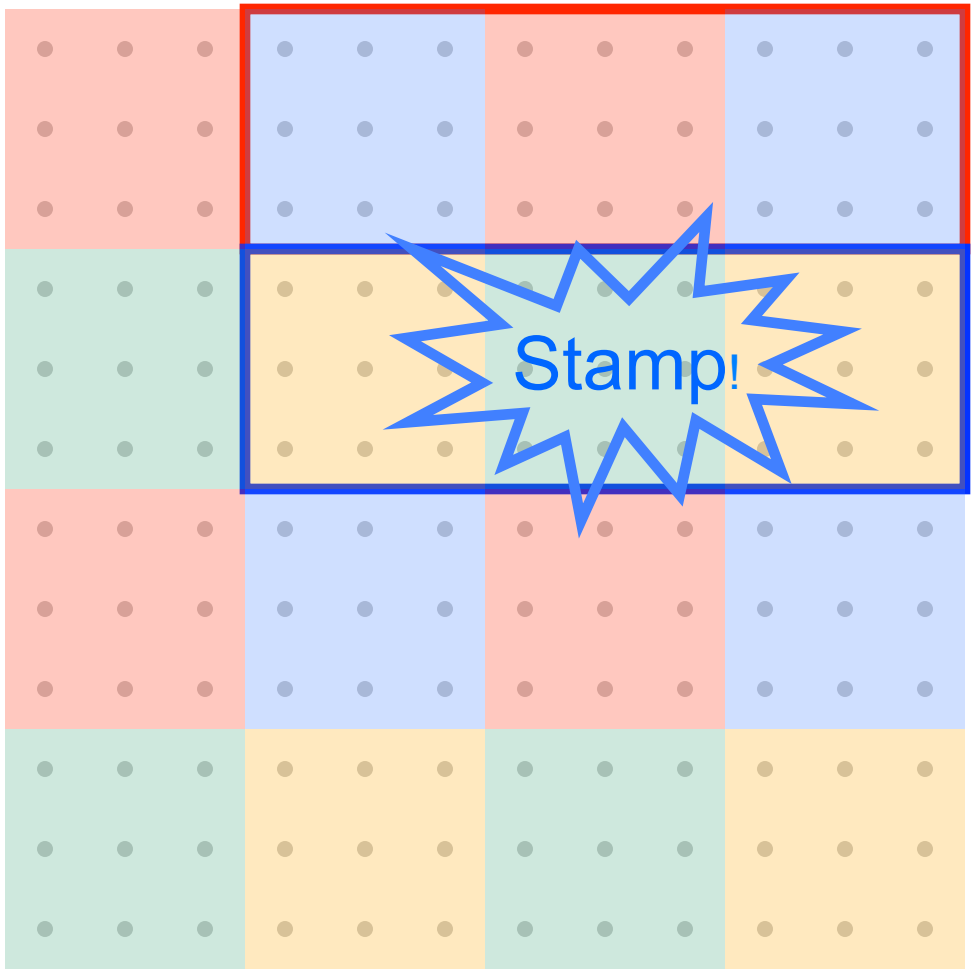
Stamping A



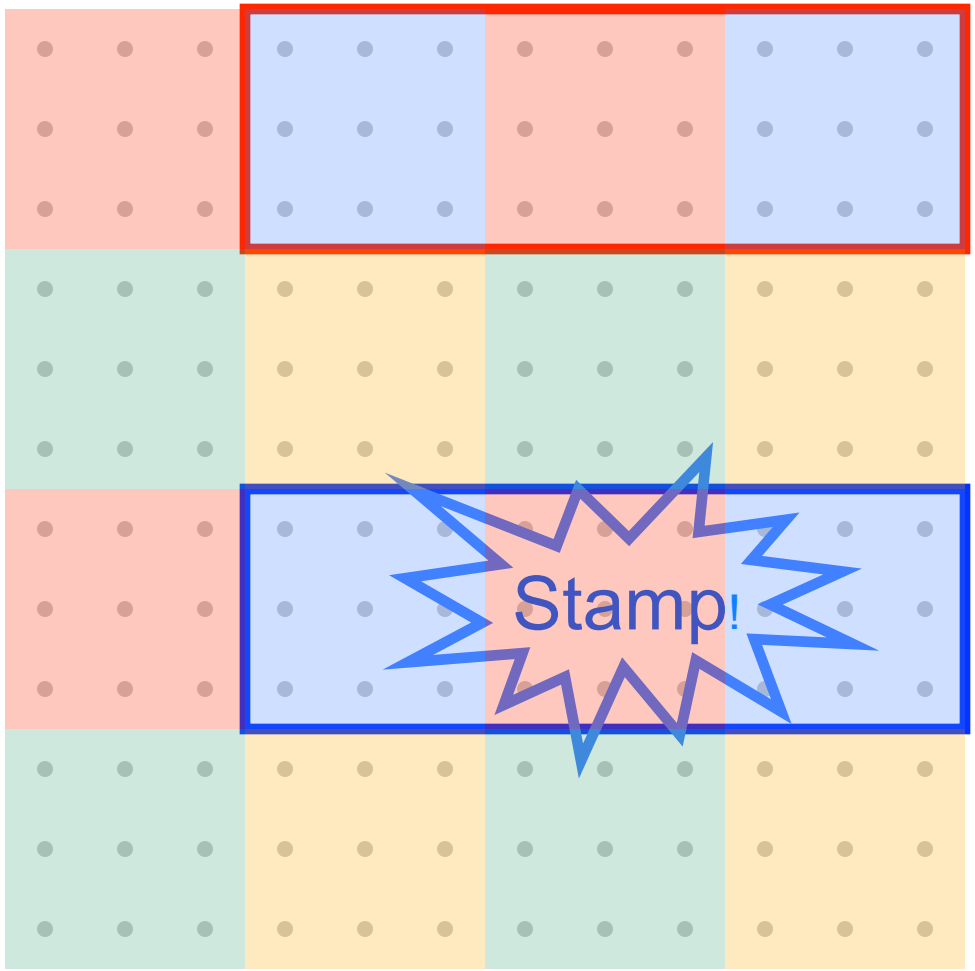
Stamping A



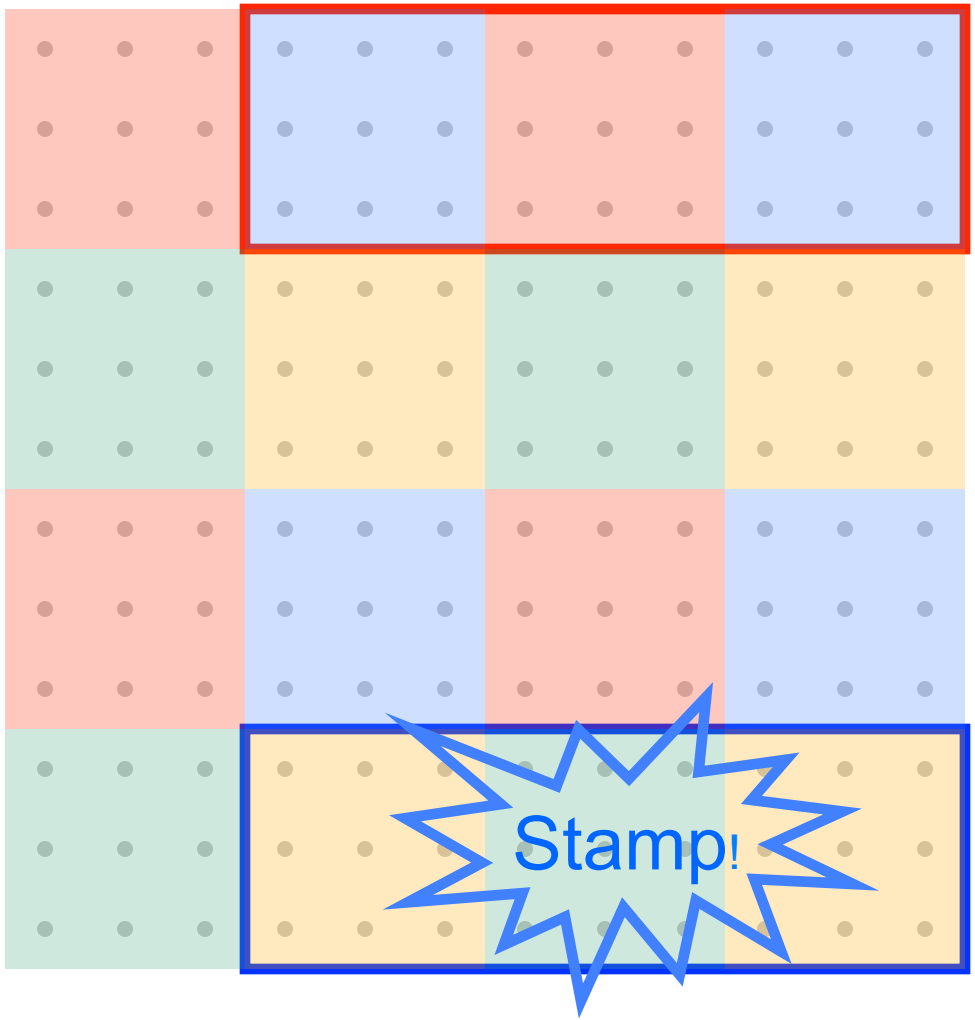
Stamping B



Stamping B

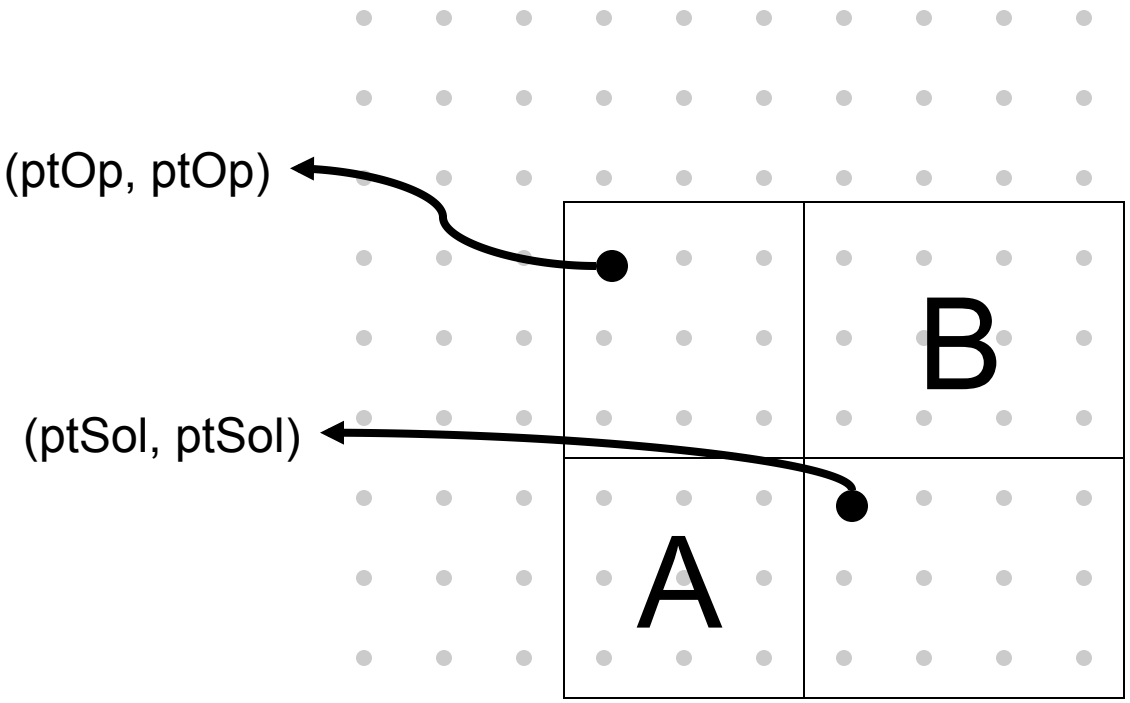


Stamping B



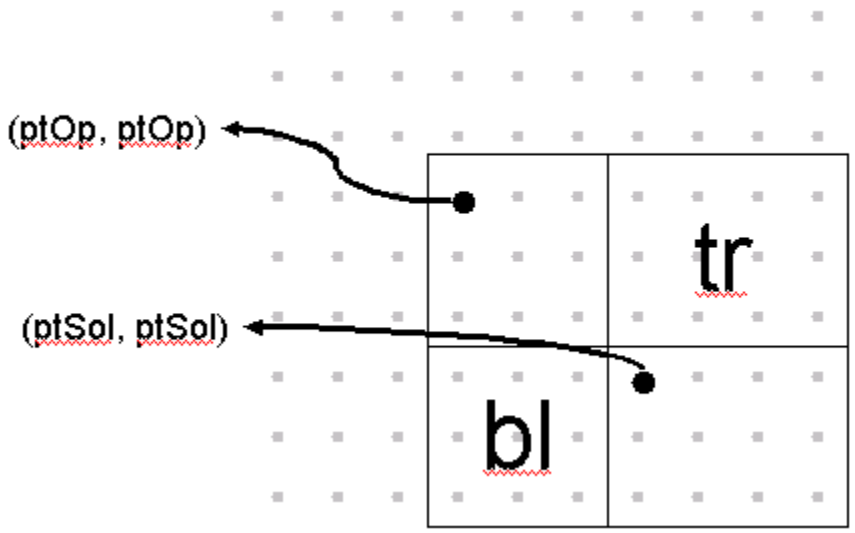
Where to copy from

```
const ASrcD = AD[ptSol.., ptOp..#blkSize];
const BSrcD = AD[ptOp..#blkSize, ptSol..];
```



Where to copy to

```
const HD = [AD.dim(1), 1..blksHoriz*blkSize];
const VD = [1..blksVert*blkSize, AD.dim(2)];

var ACopies : [HD[ptSol.., ptSol..]] real(64);
var BCopies : [VD[ptSol.., ptSol..]] real(64);
```



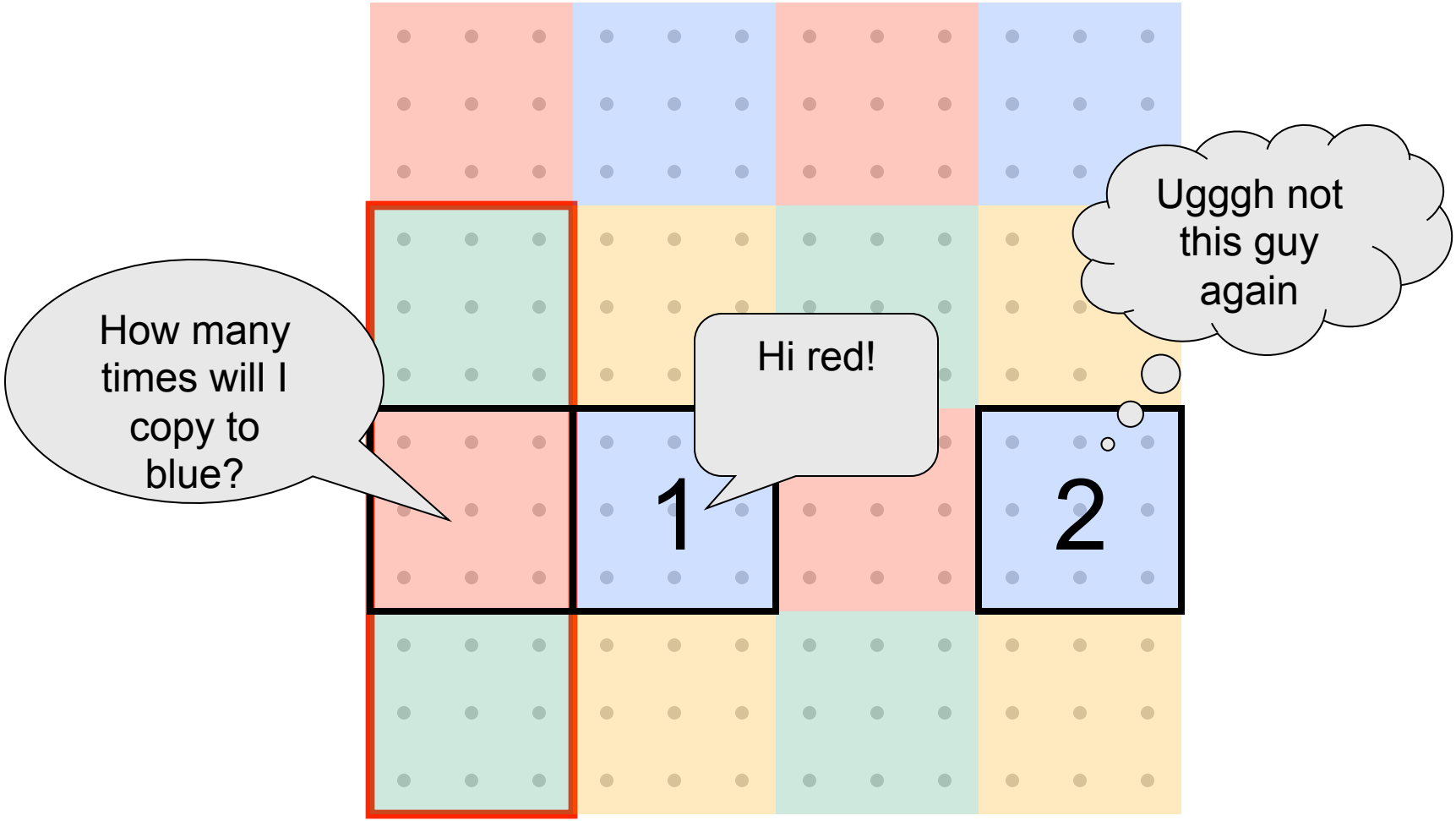
The stamping process

```
// stamp A across into ACopies. col is an index
forall col in AD.dim(2)(ptSol..) by blkSize {
    const cBlkD = HD[ptSol.., col..#blkSize];

    ACopies[cBlkD] = A[aSrcD];
}

// stamp B down into BCopies,
forall row in AD.dim(1)(ptSol..) by blkSize {
    const cBlkD = VD[row..#blkSize, ptSol..];

    BCopies[cBlkD] = A[bBlkD];
}
```

The local mults

```
// do local matrix-multiply on a block-by-block basis
forall (row,col) in AD[ptSol.., ptSol..] by (blkSize, blkSize)
{
    const ASrcD = HD[row..#blkSize, col..#blkSize];
    const BSrcD = VD[row..#blkSize, col..#blkSize];
    const CSrcD = AD[row..#blkSize, col..#blkSize];

    local {
        dgemm(
            ASrcD.dim(1).length,
            ASrcD.dim(2).length,
            BSrcD.dim(2).length,
            ACopies(ASrcD),
            BCopies(BSrcD),
            A(CSrcD));
    }
}
```

A Problem With Stamping



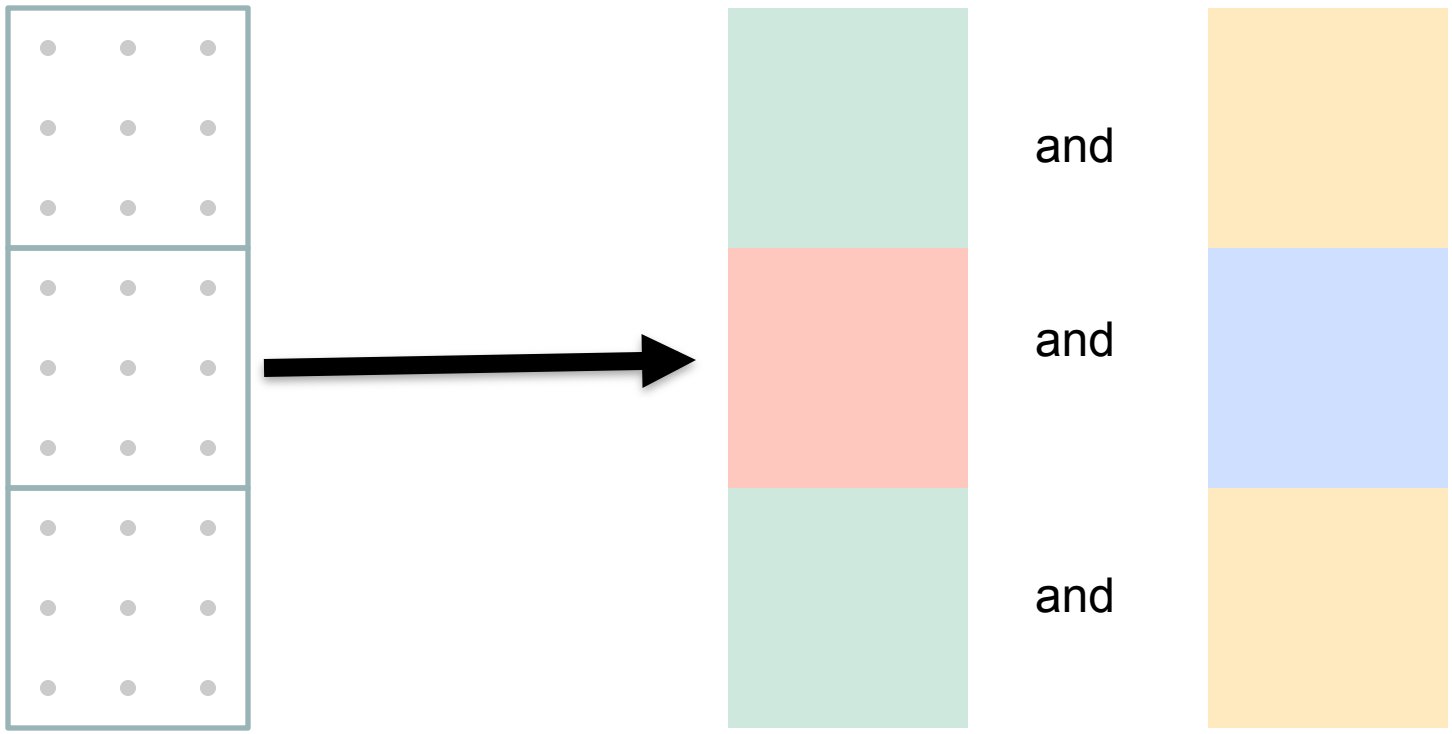
Unimplemented

Taking care of replication with a distribution

```
var AReplD = [1..n, 1..blkSize] distributed
     Dimensional(
          BlkCyc(blkSize),
          Replicated(BlkCyc(blkSize));

var ARepl : [AReplD] real(64);

ARepl = data[blk..n, blk..#blkSize];
```

How this would look

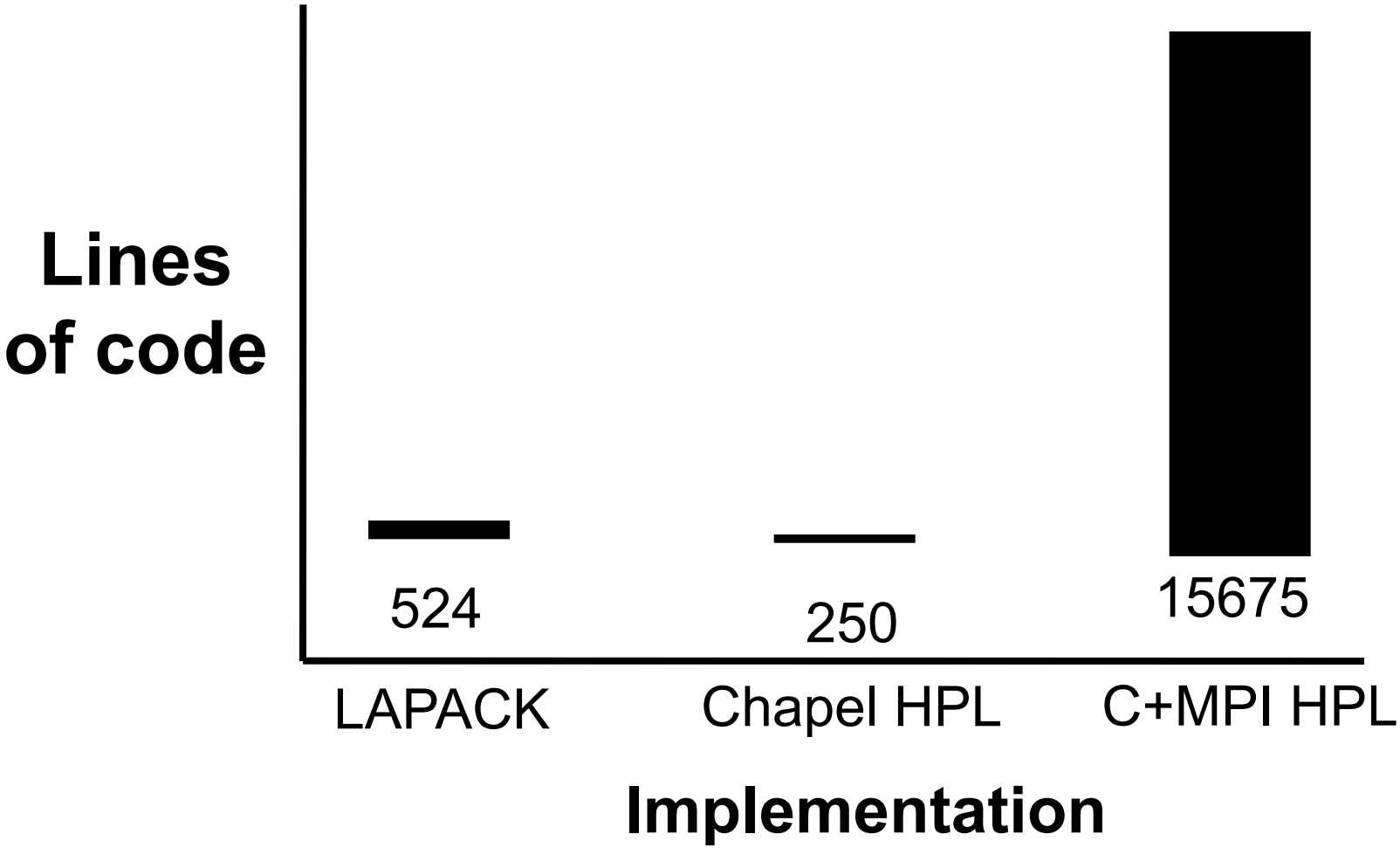
All data here gets replicated to locales:



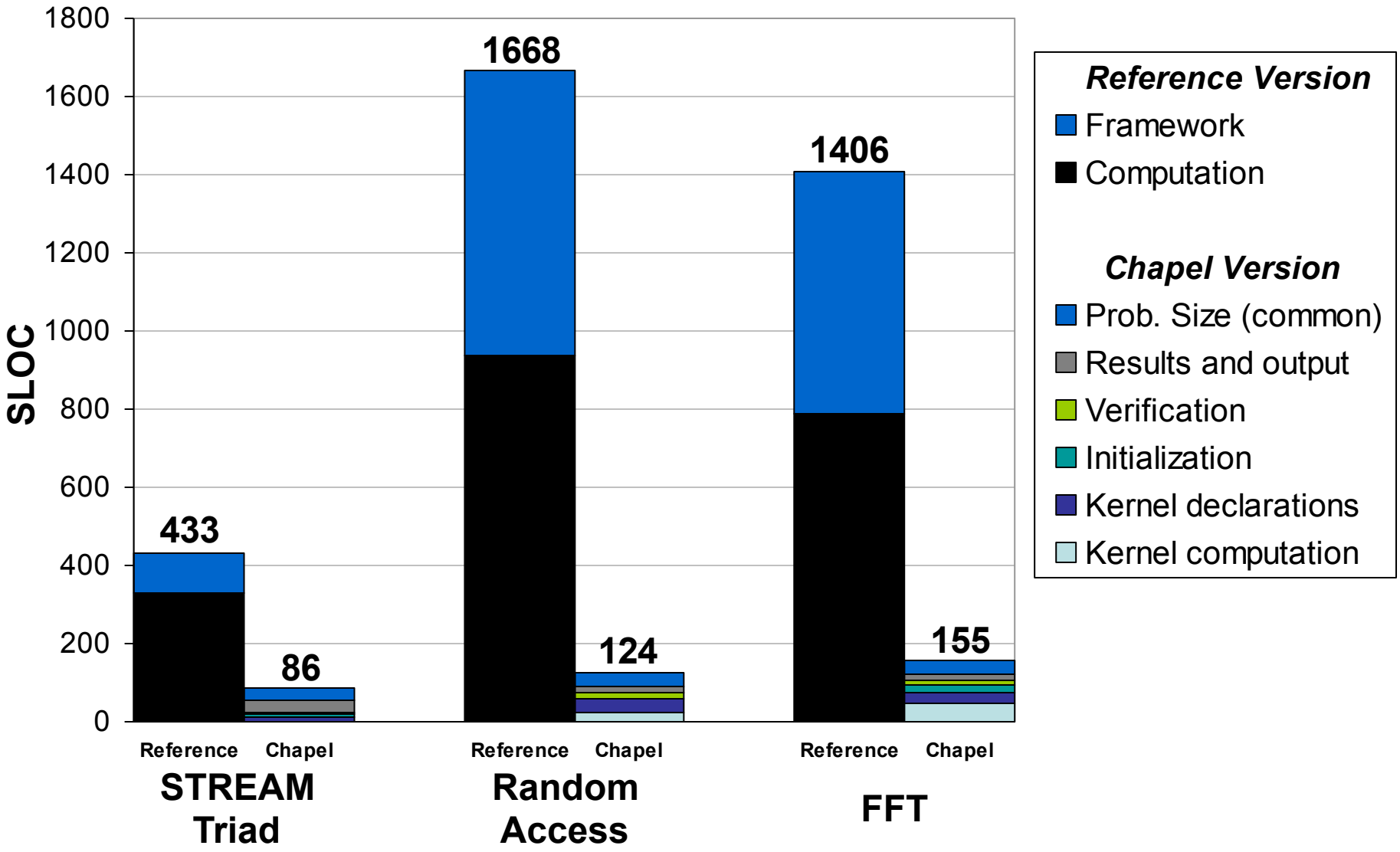
This replication happens implicitly when this array is modified

The Bottom Line

Some Numbers



What about the other benchmarks?



Does Chapel make it a breeze?

The Chapel project aims to alleviate the common hassles of parallel programming. Will it really make the arduous task of implementing HPL a breeze? Find out at this Thursday's tech forum!

Does Chapel make it a breeze?

Chapel is still very new, and in many respects unlike Fortran and C+MPI, it'll take some time to learn the best practices and idioms for development.

The programmer is still tasked with a lot – parallelism isn't free!

My code for HPL is only 250 lines long!

Chapels domain and distribution abstractions make it easier to write and read

Beating a metaphor to death

In an ideal world a person could just write the serial algorithm
(remember that code on slide 21)
that would be **a breeze**!

Programming in Chapel's really like a **moderate wind**.

But programming in C+MPI is a bit like suffering through a
class 5 hurricane without a raincoat.

Future Work

Implementation of Distributions

Performance Measurement and Optimization

Code size and expressiveness comparison with other languages

How does Chapel do in domains outside of Linear Algebra?

Learn More About Chapel

Tutorial at Super Computing 2008 in Austin

Google Seattle Conference on Scalability 2008
Lecture on YouTube
(http://www.youtube.com/watch?v=dK8IdrJrYtE)

The webpage:

http://chapel.cs.washington.edu/

Parallel Programmability and the Chapel Language

Conclusions

HPL is a well-known and important benchmark, plus the HPC Challenge requires it.

In-order to efficiently compute LU one needs to use a blocking algorithm, on each iteration of the algorithm the matrix is partitioned

Chapel's concepts of domains, and its ability to declare sub-domains by slicing, makes partitioning easy.

Chapel's distributions make the details of writing HPL's parallel matrix-multiply easier.