

# Parallel Programming and Parallel Abstractions in Fortress

**Guy Steele**

Sun Microsystems Laboratories

September 20, 2005

# “To Do for Fortran What Java™ Did for C”

- Catch “stupid mistakes”
- Extensive libraries
- Platform independence
- Security model, including type safety
- Multithreading
- Dynamic compilation
  
- Make programmers more productive

# The Context of the Research

- **Improving programmer productivity** for scientific and engineering applications
- Research funded in part by the DARPA IPTO (Defense Advanced Research Projects Agency Information Processing Technology Office) through their **High Productivity Computing Systems** program
- Goal is economically viable technologies for both government and industrial applications by the year **2010 and beyond**

# Key Ideas

- Don't build the language—grow it
- Make programming notation closer to math
- Ease use of parallelism

# Growing a Language

- Languages have gotten much bigger
- You can't build one all at once
- Therefore it must grow over time
- Might as well design it to grow
- Need to grow a user community, too
- Buy-in comes best from participation

See Steele, “Growing a Language” keynote talk, OOPSLA 1998;  
*Higher-Order and Symbolic Computation* **12**, 221–236 (1999)

# What Data Types to Include?

- Integers and floating-point (what sizes?)

# What Data Types to Include?

- Ints and floating-point (what sizes?)
- Complex numbers
- Intervals
- (Big) integers, rational numbers

# What Data Types to Include?

- Integers and floating-point (what sizes?)
- Complex numbers
- Intervals
- (Big) integers, rational numbers
- Arrays, vectors, and matrices

# What Data Types to Include?

- Integers and floating-point (what sizes?)
- Complex numbers
- Intervals
- (Big) integers, rational numbers
- Arrays, vectors, and matrices
- Rational intervals, complex intervals
- Complex vectors and matrices

# What Data Types to Include?

- Integers and floating-point (what sizes?)
- Complex numbers
- Intervals
- (Big) integers, rational numbers
- Arrays, vectors, and matrices
- Rational intervals, complex intervals
- Complex vectors and matrices
- Rational interval matrices
- Extended-float interval matrix hashtables

# What Data Types to Include?

- Integers and floating-point (what sizes?)
- Complex numbers
- Intervals
- (Big) integers, rational numbers
- Arrays, vectors, and matrices
- Rational intervals, complex intervals
- Complex vectors and matrices
- Rational interval matrices
- Extended-float interval matrix hashtables

# Minimalist Approach

- As few primitive types as possible (cf. Bacon's Kava)
- User-defined parameterized types
- User-defined polymorphic operators
- Aggressive type inference to reduce clutter
- Aggressive static and dynamic optimization
- Complex, Rational, Interval, Vector, Matrix, like Hashtable, are defined by library, coded in Fortress
  
- BUT: we want nice notation, not `x.multiply(y)`
- These are existing techniques—let's put 'em to work

# Interesting Language Design Strategy

Wherever possible,  
consider whether a proposed language feature  
can be provided by a library  
rather than having it wired into the compiler.

# A Growable, Open Language

- Push decisions out to libraries
- Old language design model:
  - > Study applications
  - > Add language features to improve application coding
- Our new model:
  - > Study applications
  - > Study how a library can improve application coding
  - > Add language features to improve library coding
- Conjectures:
  - > Better leverage, leading to more rapid improvement
  - > Enables experimentation with open-source strategies

# Making Abstraction Efficient

- We assume implementation technology that makes aggressive use of runtime performance measurement and optimization.
- Repeat the success of the Java™ Virtual Machine
- Goal: programmers (especially library writers) need not fear subroutines, functions, methods, and interfaces for performance reasons
- This may take years, but we're talking 2010

# Core Language Is a Framework

- Very general, open-ended syntax
- Lots of support for abstraction
- Powerful polymorphic, parameterized type system
  - > Make certain kinds of flow analysis explicit
- Fortress coded as a set of libraries in this framework
- Thus we hope to support other domain-specific languages or sublanguages

# Type System: Objects and Traits

- Traits: like interfaces, but may contain code
  - > Based on work by Schärli, Ducasse, Nierstrasz, Black, et al.
- Multiple inheritance of code (but not fields)
- Traits and methods may be parameterized
  - > Parameters may be traits or compile-time constants
- Primitive types are first-class
  - > Booleans, integers, floats, characters are all objects

# Data and Control Models

- Data model: shared global address space
- Control model: multithreaded
  - > Basic primitive is “spawn”
- Transactional access to shared variables
  - > Atomic blocks
  - > Explicit testing and signaling of failure/retry
  - > Lock-free (no blocking, no deadlock)
  - > Cf. work by Herlihy, Moss, and others on transactional memory

# Should Parallelism Be the Default?

- “Loop” can be a misleading term
  - > A set of executions of a parameterized block of code
  - > Whether to order or parallelize those executions should be a separate question
  - > Maybe you should have to ask for sequential execution!
- Fortress “loops” are parallel by default
  - > This is actually a library convention about generators

# Parallelism Is the Default

```
for i←1:m, j←1:n do a[i,j] := b[i] c[j] end
```

```
for i←seq(1:m) do
  for j←seq(1:n) do
    print a[i,j]
  end
end
```

```
for (i,j)←a.indices do a[i,j] := b[i] c[j] end
```

- Generators (defined by libraries) manage parallelism and the assignment of threads to processors

```
y = Σ[k←1:n] a[k] x^k
```

```
z = MAX[(i,j)←a.indices] a[i,j]
```

- Reducers (also defined by libraries) such as  $\Sigma$  (or **SUM**) and **MAX** may have serial/parallel implementations

# Conventional Mathematical Notation

- The language of mathematics is centuries old, concise, convenient, and widely taught
- Programming language notation can become closer to mathematical notation (Unicode helps a lot)
  - >  $\mathbf{v\_norm = v / \|v\|}$
  - >  $\mathbf{\Sigma[k=1:n] a[k] x^k}$
  - >  $\mathbf{C = A \cup B}$
  - >  $\mathbf{y = 3 x \sin x \cos 2 x \log \log x}$
- Parsing this stuff is an interesting research problem

# What Syntax is Actually Wired In?

- Parentheses ( ) for grouping
- Comma , to separate expressions in tuples
- Semicolon ; to separate statements on a line
- Dot . for field and method selection
- Juxtaposition is a binary operator
- Any other operator can be infix, prefix, and/or postfix
- Many sets of brackets
- Conservative, traditional rules of precedence
  - > A dag, not always transitive (examples:  $A+B>C$  is okay; so is  $B>C \vee D>E$ ; but  $A+B \vee C$  needs parentheses)

# Libraries Define . . .

- Which operators have infix, prefix, postfix definitions, and what types they apply to
  - opr**  $-(m:\text{Integer}, n:\text{Integer}) = m.\text{subtract}(n)$
  - opr**  $-(m:\text{Integer}) = m.\text{negate}()$
  - opr**  $(n:\text{Integer})! = \text{if } n=0 \text{ then } 1 \text{ else } (n-1)! \text{ end}$
- Whether a juxtaposition is meaningful
  - opr**  $\text{juxta}(m:\text{Integer}, n:\text{Integer}) = m.\text{times}(n)$
- What bracketing operators actually mean
  - opr**  $\lceil x:\text{Number} \rceil = \text{ceiling}(x)$
  - opr**  $|x:\text{Number}| = \text{if } x < 0 \text{ then } -x \text{ else } x \text{ end}$
  - opr**  $|s:\text{Set}| = s.\text{size}$

# But Wasn't Operator Overloading a Disaster in C++ ?

- Yes, it was
  - > Not enough operators to go around
  - > Failure to stick to traditional meanings
- We have also been tempted and had to resist
- We see benefits in using notations for programming that are also used for specification

# Simple Example: NAS CG Kernel (ASCII)

```
conjGrad(A: Matrix[/Float/], x: Vector[/Float/]):
  (Vector[/Float/], Float)
```

```
  cgit_max = 25
  z: Vector[/Float/] = 0
  r: Vector[/Float/] = x
  p: Vector[/Float/] = r
  rho: Float = r^T r
  for j <- seq(1:cgit_max) do
    q = A p
    alpha = rho / p^T q
    z := z + alpha p
    r := r - alpha q
    rho0 = rho
    rho := r^T r
    beta = rho / rho0
    p := r + beta p
  end
  (z, ||x - A z||)
```

```
(z, norm) = conjGrad(A, x)
```

Matrix[/T/] and Vector[/T/] are parameterized interfaces, where T is the type of the elements.

The form  $x:T=e$  declares a variable x of type T with initial value e, and that variable may be updated using the assignment operator  $:=$ .

# Simple Example: NAS CG Kernel (ASCII)

```

conjGrad[/Elt extends Number, nat N,
         Mat extends Matrix[/Elt,N BY N/],
         Vec extends Vector[/Elt,N/]
        /](A: Mat, x: Vec): (Vec, Elt)

```

```

cgitmax = 25
z: Vec = 0
r: Vec = x
p: Vec = r
rho: Elt = r^T r
for j <- seq(1:cgit_max) do
  q = A p
  alpha = rho / p^T q
  z := z + alpha p
  r := r - alpha q
  rho0 = rho
  rho := r^T r
  beta = rho / rho0
  p := r + beta p
end
(z, ||x - A z||)

```

Here we make conjGrad a generic procedure. The runtime compiler may produce multiple instantiations of the code for various types E.

The form  $x=e$  as a statement declares variable  $x$  to have an unchanging value. The type of  $x$  is exactly the type of the expression  $e$ .

```
(z,norm) = conjGrad(A,x)
```

# Simple Example: NAS CG Kernel (Unicode)

```

conjGrad[[Elt extends Number, nat N,
         Mat extends Matrix[[Elt,N×N]],
         Vec extends Vector[[Elt,N]]
         ]](A: Mat, x: Vec): (Vec, Elt)
  cgit_max = 25
  z: Vec = 0
  r: Vec = x
  p: Vec = r
  ρ: E = r^T r
  do j ← seq(1:cgit_max) do
    q = A p
    α = ρ / p^T q
    z := z + α p
    r := r - α q
    ρ₀ = ρ
    ρ := r^T r
    β = ρ / ρ₀
    p := r + β p
  end do
  return (z, ||x - A z||)

```

This would be considered entirely equivalent to the previous version. You might think of this as an abbreviated form of the ASCII version, or you might think of the ASCII version as a way to conveniently enter this version on a standard keyboard.

# Simple Example: NAS CG Kernel

```

conjGrad  $\llbracket$  Elt extends Number, nat N,
           Mat extends Matrix  $\llbracket$  Elt,  $N \times N$   $\rrbracket$ ,
           Vec extends Vector  $\llbracket$  Elt, N  $\rrbracket$ 
            $\rrbracket$  ( $A$ : Mat,  $x$ : Vec): (Vec, Elt)
  
```

```
cgmax = 25
```

```
 $z$ : Vec = 0
```

```
 $r$ : Vec =  $x$ 
```

```
 $p$ : Vec =  $r$ 
```

```
 $\rho$ : Elt =  $r^T r$ 
```

```
for  $j \leftarrow$  seq(1: cgmax) do
```

```
   $q = A p$ 
```

```
   $\alpha = \frac{\rho}{p^T q}$ 
```

```
   $z := z + \alpha p$ 
```

```
   $r := r - \alpha q$ 
```

```
   $\rho_0 = \rho$ 
```

```
   $\rho := r^T r$ 
```

```
   $\beta = \frac{\rho}{\rho_0}$ 
```

```
   $p := r + \beta p$ 
```

```
end
```

```
( $z$ ,  $\|x - A z\|$ )
```

It's not new or surprising that code written in a programming language might be displayed in a conventional math-like format. The point of this example is how similar the code is to the math notation: the gap between the two syntaxes is relatively small. We want to see what will happen if a principal goal of a new language design is to minimize this gap.

# Comparison: NAS NPB 1 Specification

```

z = 0
r = x
ρ = rT r
p = r
DO i = 1, 25
  q = A p
  α = ρ / (pT q)
  z = z + α p
  ρ0 = ρ
  r = r - α q
  ρ = rT r
  β = ρ / ρ0
  p = r + β p
ENDDO
compute residual norm explicitly: ||r|| = ||x - A z||

```

```

z : Vec = 0
r : Vec = x
p : Vec = r
ρ : Elt = rT r
for j ← seq(1 : cgitmax) do
  q = A p
  α =  $\frac{\rho}{p^T q}$ 
  z := z + α p
  r := r - α q
  ρ0 = ρ
  ρ := rT r
  β =  $\frac{\rho}{\rho_0}$ 
  p := r + β p
end
(z, ||x - A z||)

```

# Comparison: NAS NPB 2.3 Serial Code

```

do j=1,naa+1
  q(j) = 0.0d0
  z(j) = 0.0d0
  r(j) = x(j)
  p(j) = r(j)
  w(j) = 0.0d0
enddo
sum = 0.0d0
do j=1,lastcol-firstcol+1
  sum = sum + r(j)*r(j)
enddo
rho = sum
do cgit = 1,cgitmax
  do j=1,lastrow-firstrow+1
    sum = 0.d0
    do k=rowstr(j),rowstr(j+1)-1
      sum = sum + a(k)*p(colidx(k))
    enddo
    w(j) = sum
  enddo
  do j=1,lastcol-firstcol+1
    q(j) = w(j)
  enddo
enddo

do j=1,lastcol-firstcol+1
  w(j) = 0.0d0
enddo
sum = 0.0d0
do j=1,lastcol-firstcol+1
  sum = sum + p(j)*q(j)
enddo
d = sum
alpha = rho / d
rho0 = rho
do j=1,lastcol-firstcol+1
  z(j) = z(j) + alpha*p(j)
  r(j) = r(j) - alpha*q(j)
enddo
sum = 0.0d0
do j=1,lastcol-firstcol+1
  sum = sum + r(j)*r(j)
enddo
rho = sum
beta = rho / rho0
do j=1,lastcol-firstcol+1
  p(j) = r(j) + beta*p(j)
enddo
enddo

do j=1,lastrow-firstrow+1
  sum = 0.d0
  do k=rowstr(j),rowstr(j+1)-1
    sum = sum + a(k)*z(colidx(k))
  enddo
  w(j) = sum
enddo
do j=1,lastcol-firstcol+1
  r(j) = w(j)
enddo
sum = 0.0d0
do j=1,lastcol-firstcol+1
  d = x(j) - r(j)
  sum = sum + d*d
enddo
d = sum
rnorm = sqrt( d )

```

Parallel version is similar, but with MPI calls interspersed.

# Conjugate Gradient Calculation

$z = 0$

$r = x$

$\rho = r^T r$  dot product

$p = r$

**DO**  $i = 1, 25$

$q = A p$  sparse matrix-vector product

$\alpha = \rho / (p^T q)$  dot product, scalar division

$z = z + \alpha p$  daxpy

$\rho_0 = \rho$

$r = r - \alpha q$  daxpy

$\rho = r^T r$  dot product

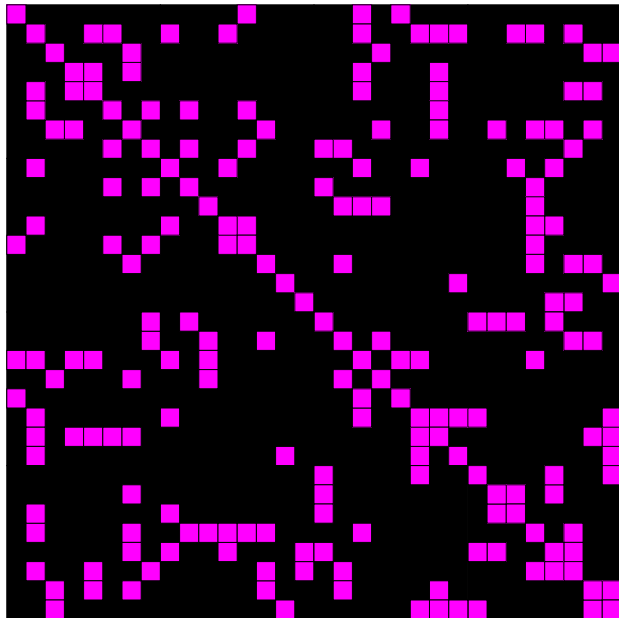
$\beta = \rho / \rho_0$  scalar division

$p = r + \beta p$  daxpy

**ENDDO**

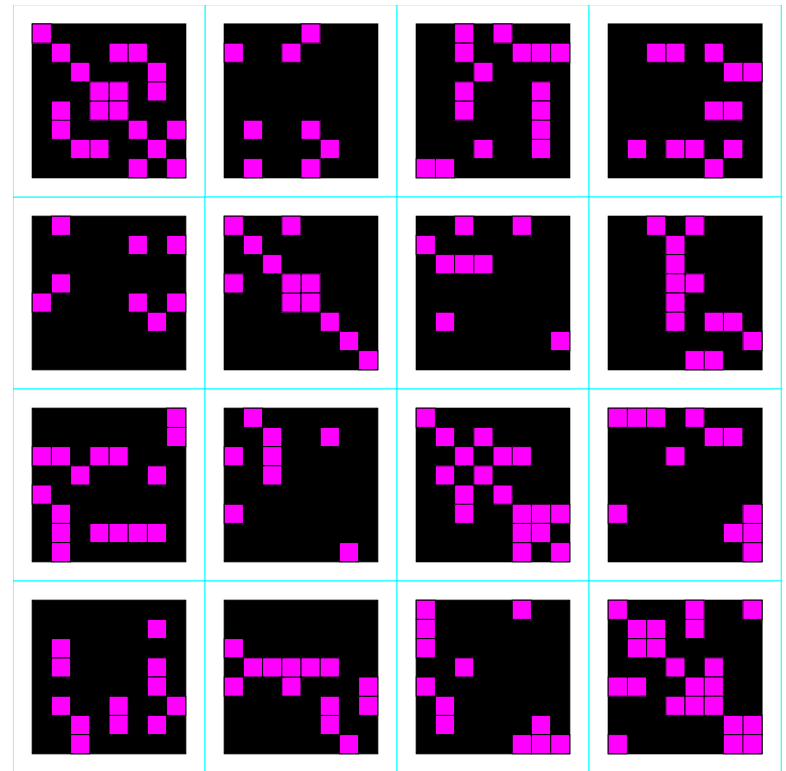
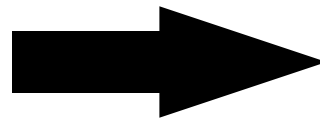
$\| r \| = \| x - A z \|$  sparse matrix-vector product, dot product

# Tiling of Sparse Matrix



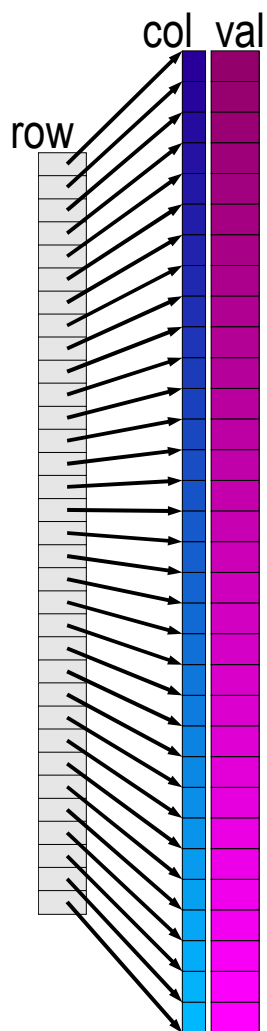
■ = nonzero entry

The matrix is tiled, each tile having (roughly) the same number of rows and columns.

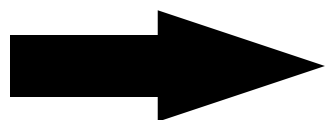


Each MPI process gets a tile.

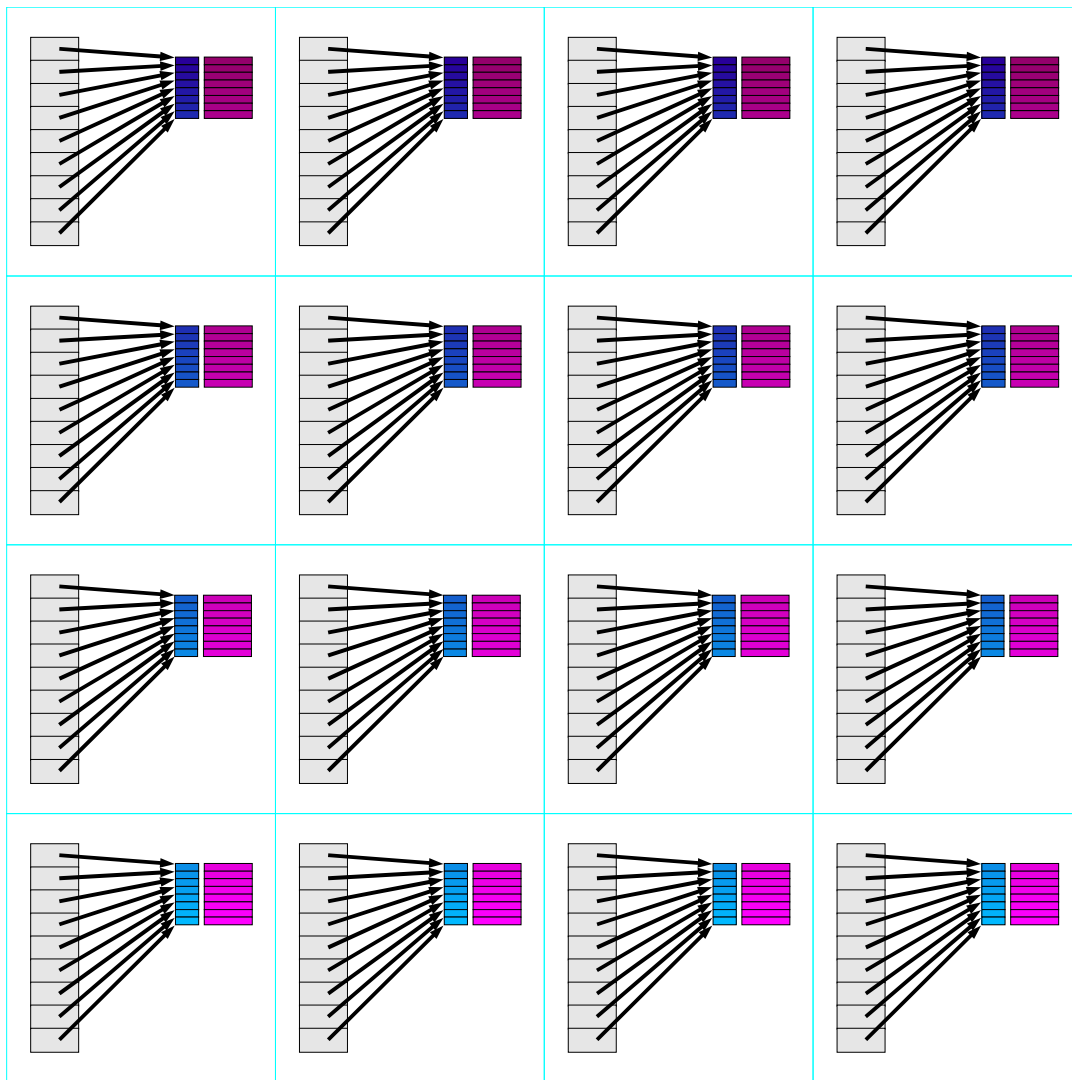
# Tiling of Sparse Matrix Representation



The matrix is tiled, each tile having (roughly) the same number of rows and columns.



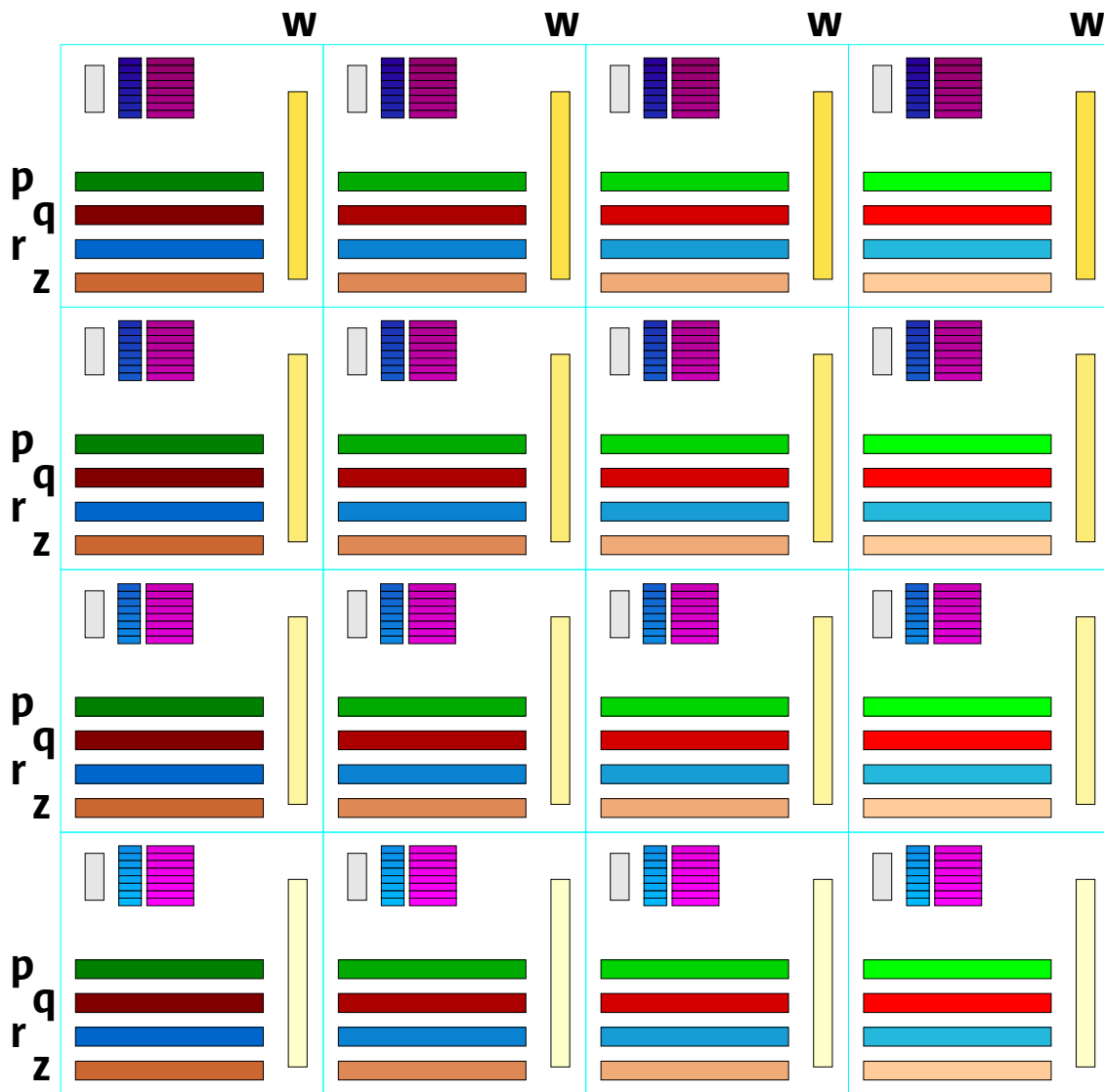
Each MPI process gets a tile.



# Representation and Alignment of Vectors

Vectors **p**, **q**, **r**, and **z** are represented in a replicated, transposed form, aligned with rows of the matrix **A**.

Temporary vector **w** is replicated but not transposed; it is aligned columnwise.



# Execution: Dot Product (1 of 3)

$z = 0$

$r = x$

$\rho = r^T r$

$p = r$

**DO**  $i = 1, 25$

$q = A p$

$\alpha = \rho / (p^T q)$

$z = z + \alpha p$

$\rho_0 = \rho$

$r = r - \alpha q$

$\rho = r^T r$

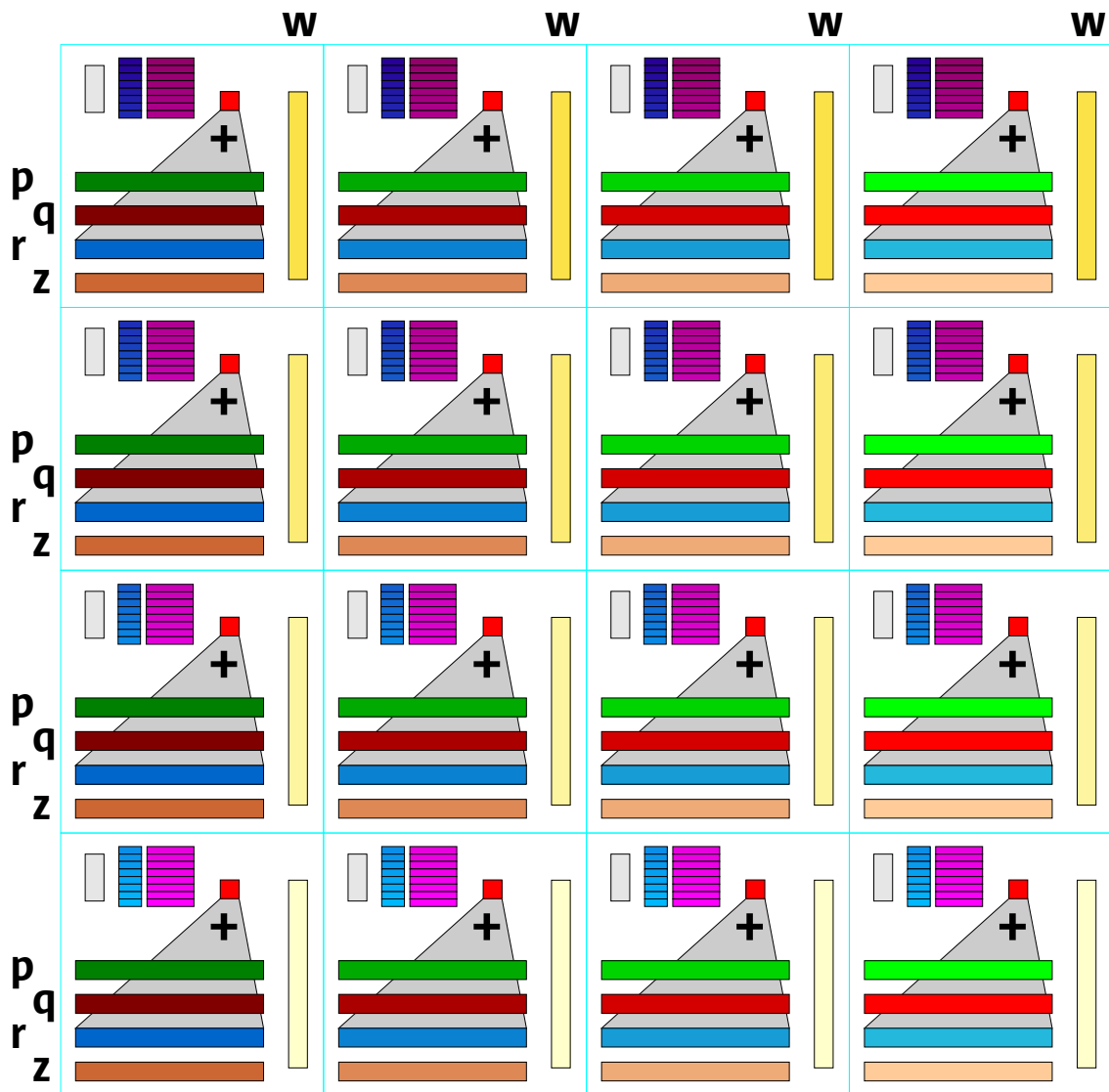
$\beta = \rho / \rho_0$

$p = r + \beta p$

**ENDDO**

$\| r \| = \| x - A z \|$

sum locally

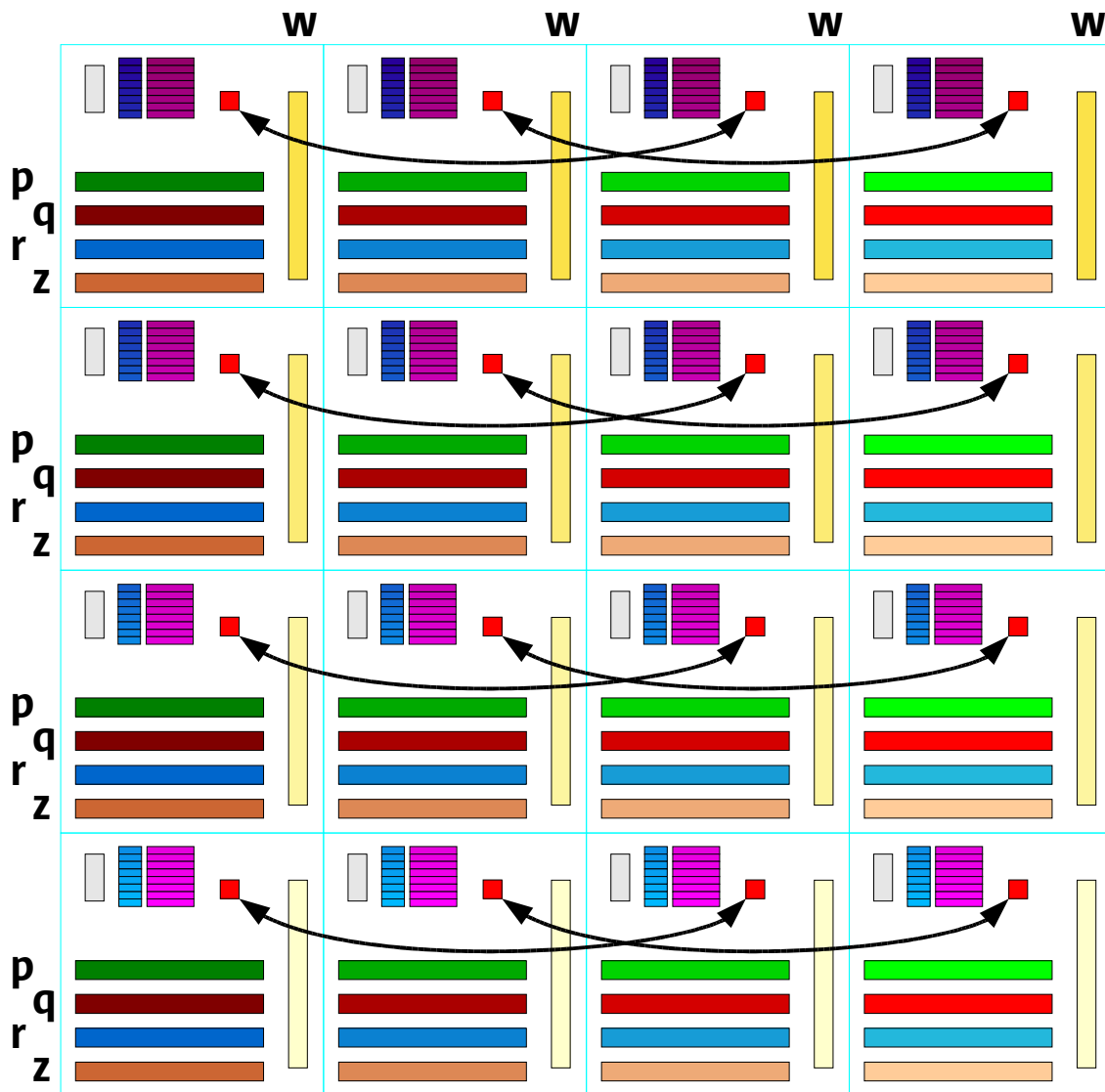


# Execution: Dot Product (2 of 3)

```

z = 0
r = x
ρ = rT r
p = r
DO i = 1, 25
  q = A p
  α = ρ / (pT q)
  z = z + α p
  ρ0 = ρ
  r = r - α q
  ρ = rT r
  β = ρ / ρ0
  p = r + β p
ENDDO
|| r || = || x - A z ||
    
```

swap and add

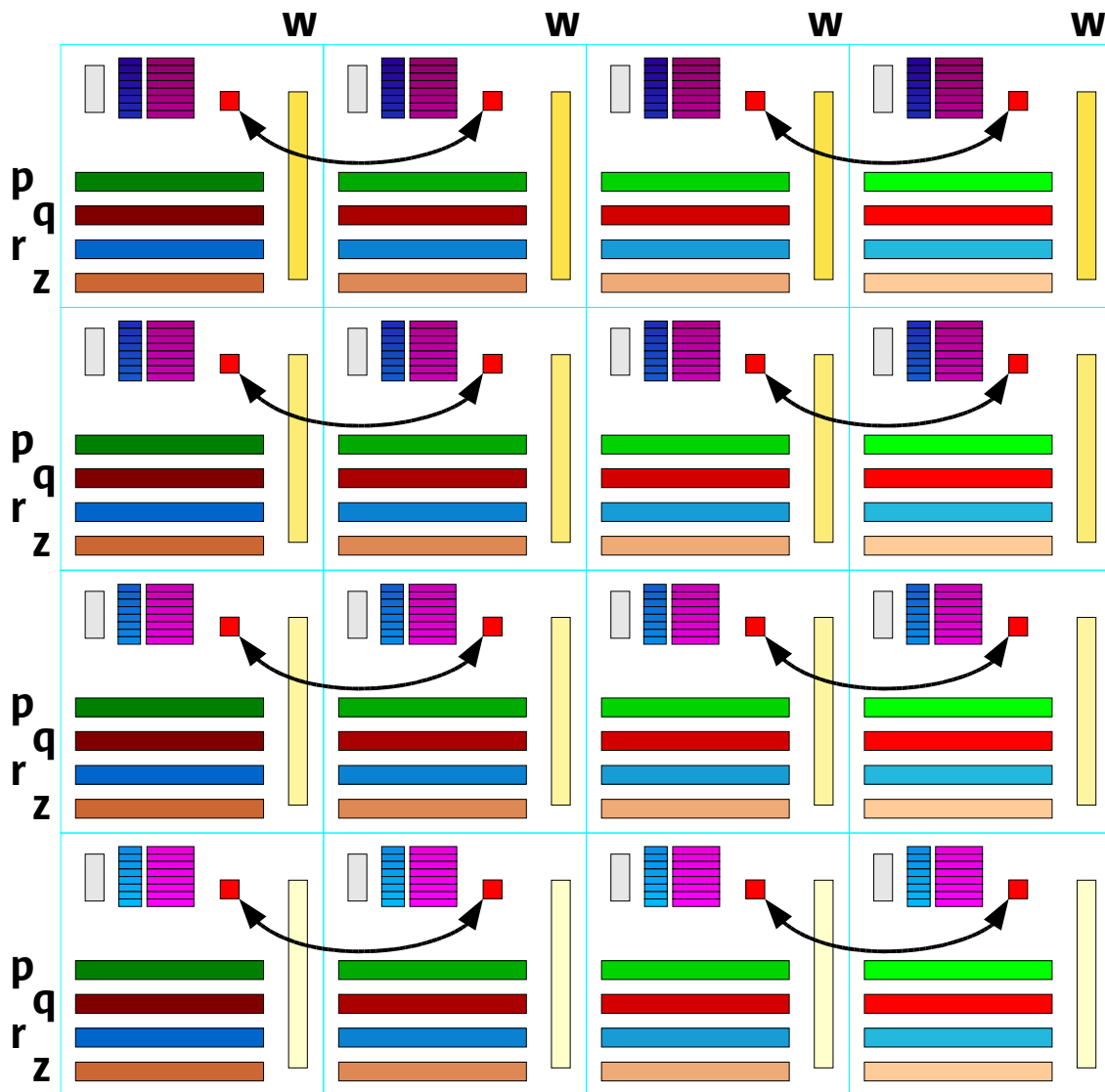


# Execution: Dot Product (3 of 3)

```

z = 0
r = x
ρ = rT r
p = r
DO i = 1, 25
  q = A p
  α = ρ / (pT q)
  z = z + α p
  ρ0 = ρ
  r = r - α q
  ρ = rT r
  β = ρ / ρ0
  p = r + β p
ENDDO
|| r || = || x - A z ||
    
```

swap and add



# Execution: Matrix-Vector Product (1 of 4)

$z = 0$       local matrix-  
 $r = x$       vector products

$\rho = r^T r$

$p = r$

**DO**  $i = 1, 25$

$q = A p$

$\alpha = \rho / (p^T q)$

$z = z + \alpha p$

$\rho_0 = \rho$

$r = r - \alpha q$

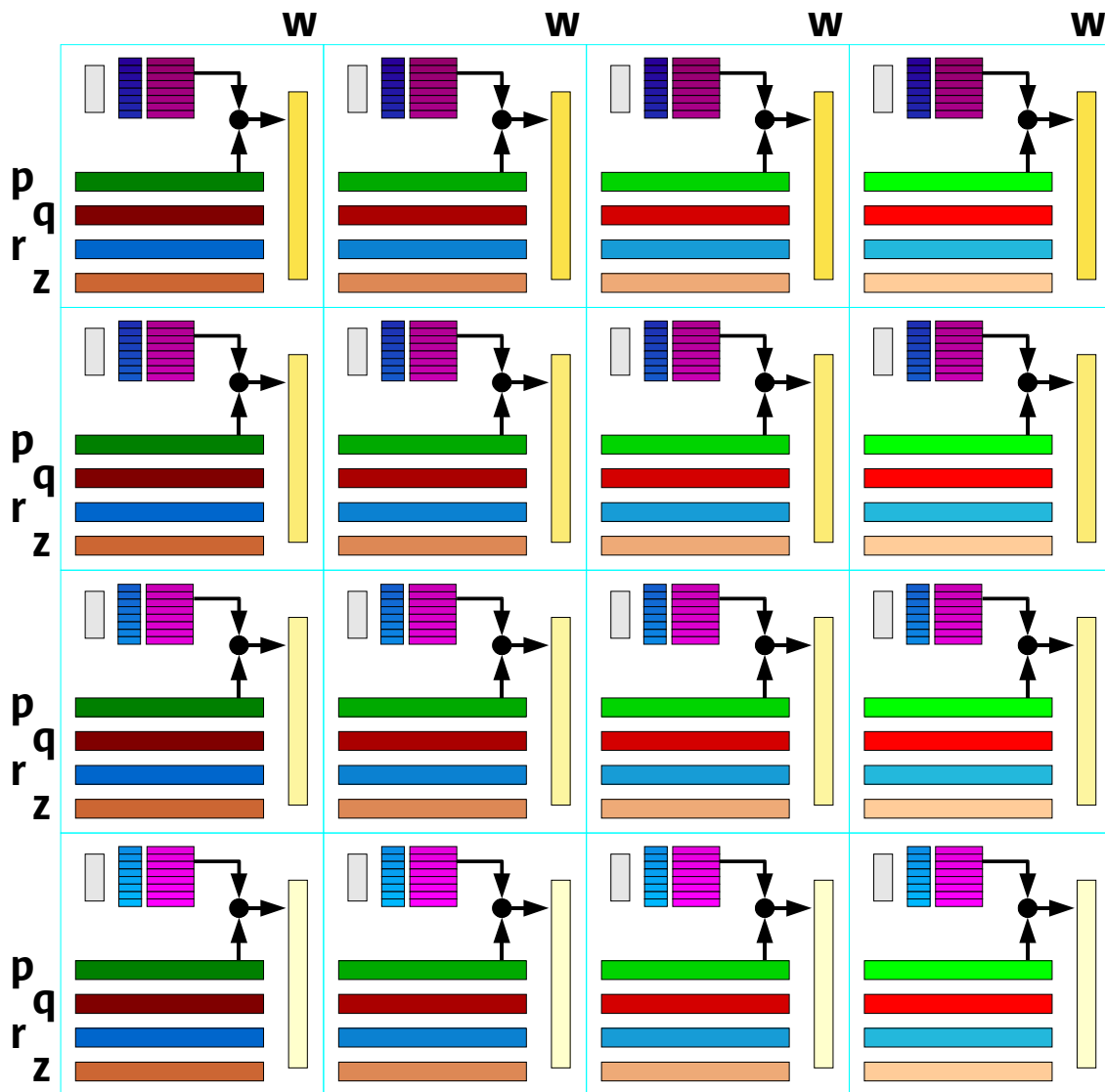
$\rho = r^T r$

$\beta = \rho / \rho_0$

$p = r + \beta p$

**ENDDO**

$\| r \| = \| x - A z \|$



# Execution: Matrix-Vector Product (2 of 4)

$z = 0$       swap and add  
 $r = x$       (elementwise)

$\rho = r^T r$

$p = r$

**DO**  $i = 1, 25$

$q = A p$

$\alpha = \rho / (p^T q)$

$z = z + \alpha p$

$\rho_0 = \rho$

$r = r - \alpha q$

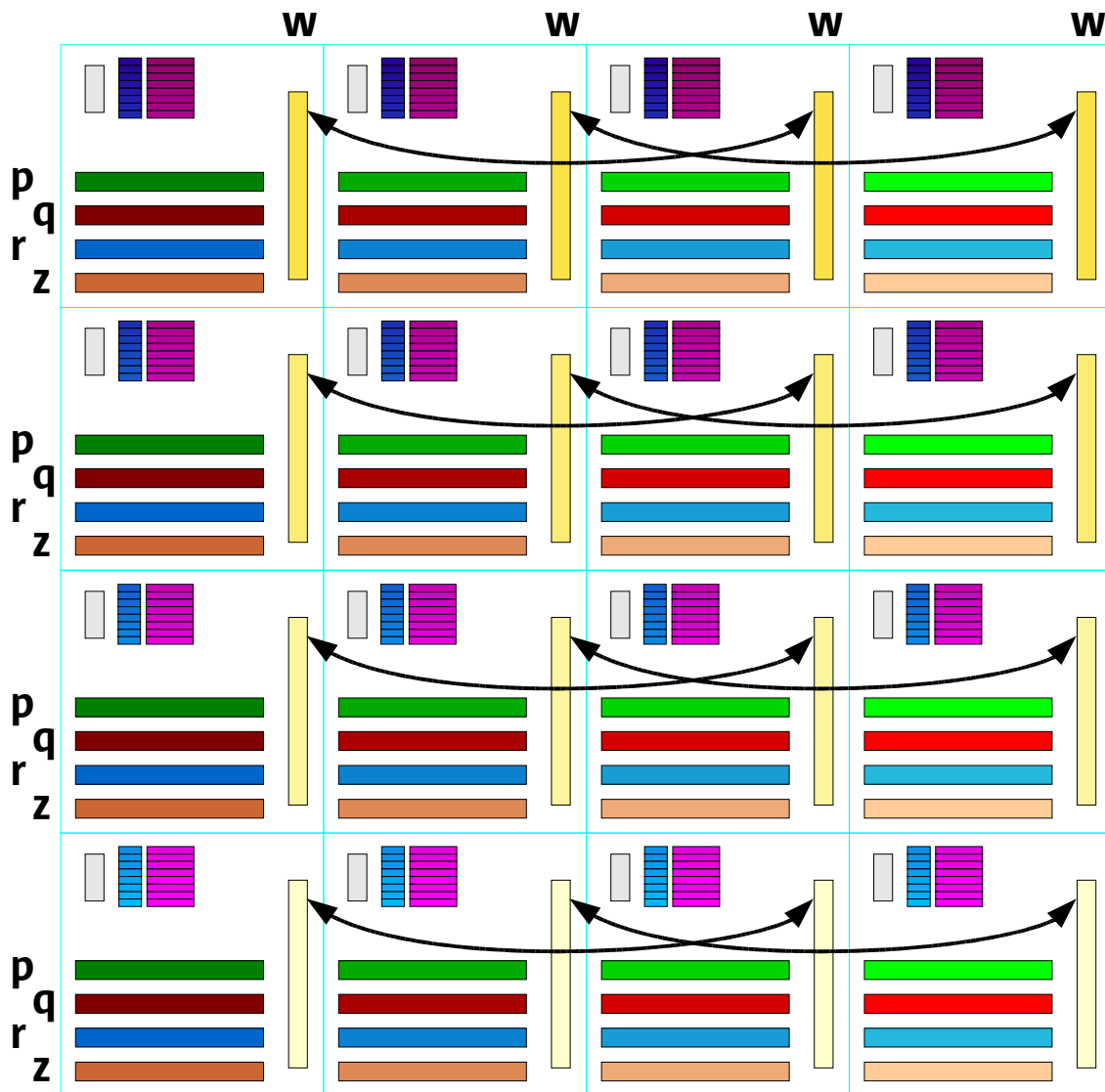
$\rho = r^T r$

$\beta = \rho / \rho_0$

$p = r + \beta p$

**ENDDO**

$\| r \| = \| x - A z \|$



# Execution: Matrix-Vector Product (3 of 4)

$z = 0$  swap and add  
 $r = x$  (elementwise)

$\rho = r^T r$

$p = r$

**DO**  $i = 1, 25$

$q = A p$

$\alpha = \rho / (p^T q)$

$z = z + \alpha p$

$\rho_0 = \rho$

$r = r - \alpha q$

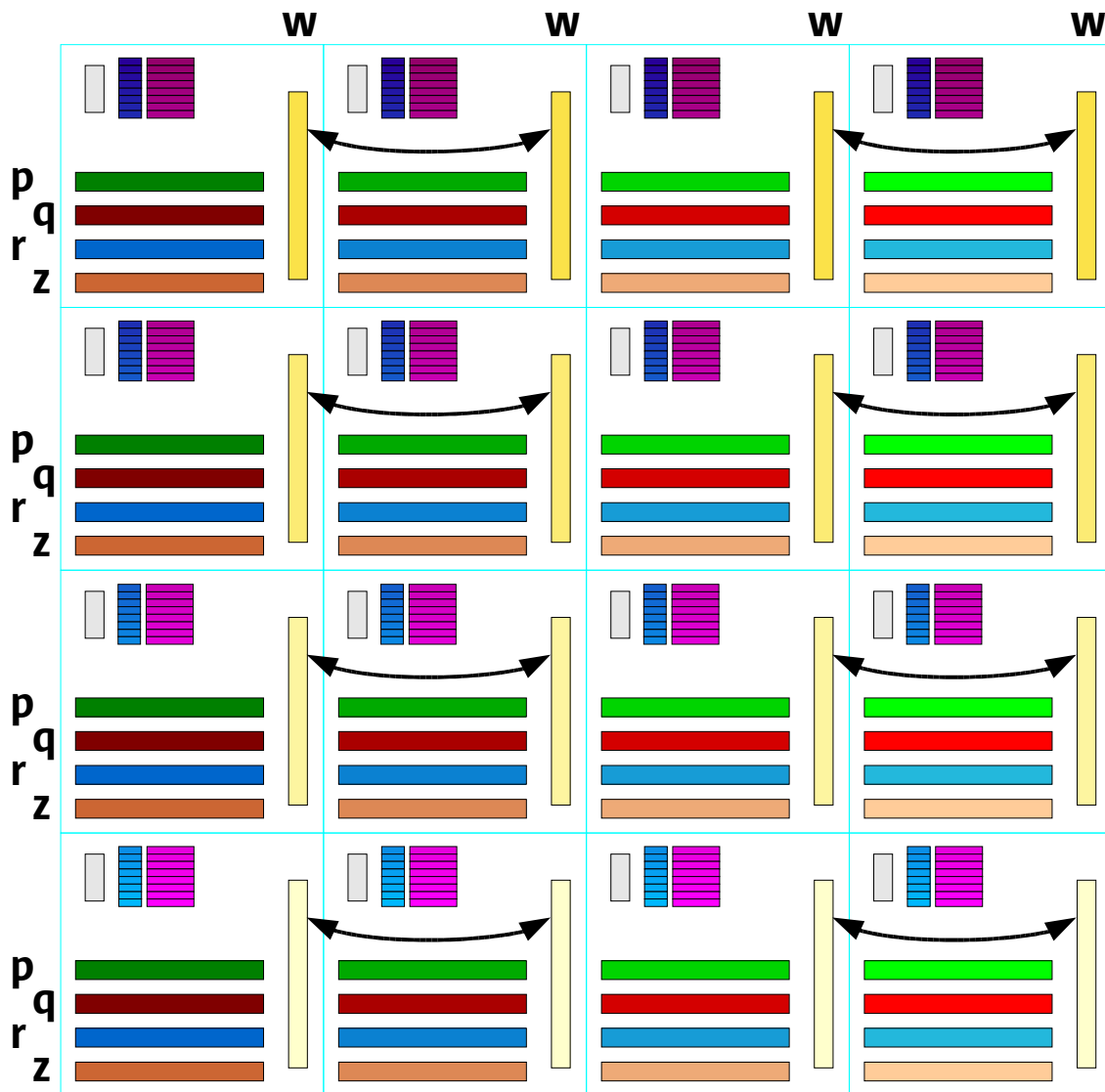
$\rho = r^T r$

$\beta = \rho / \rho_0$

$p = r + \beta p$

**ENDDO**

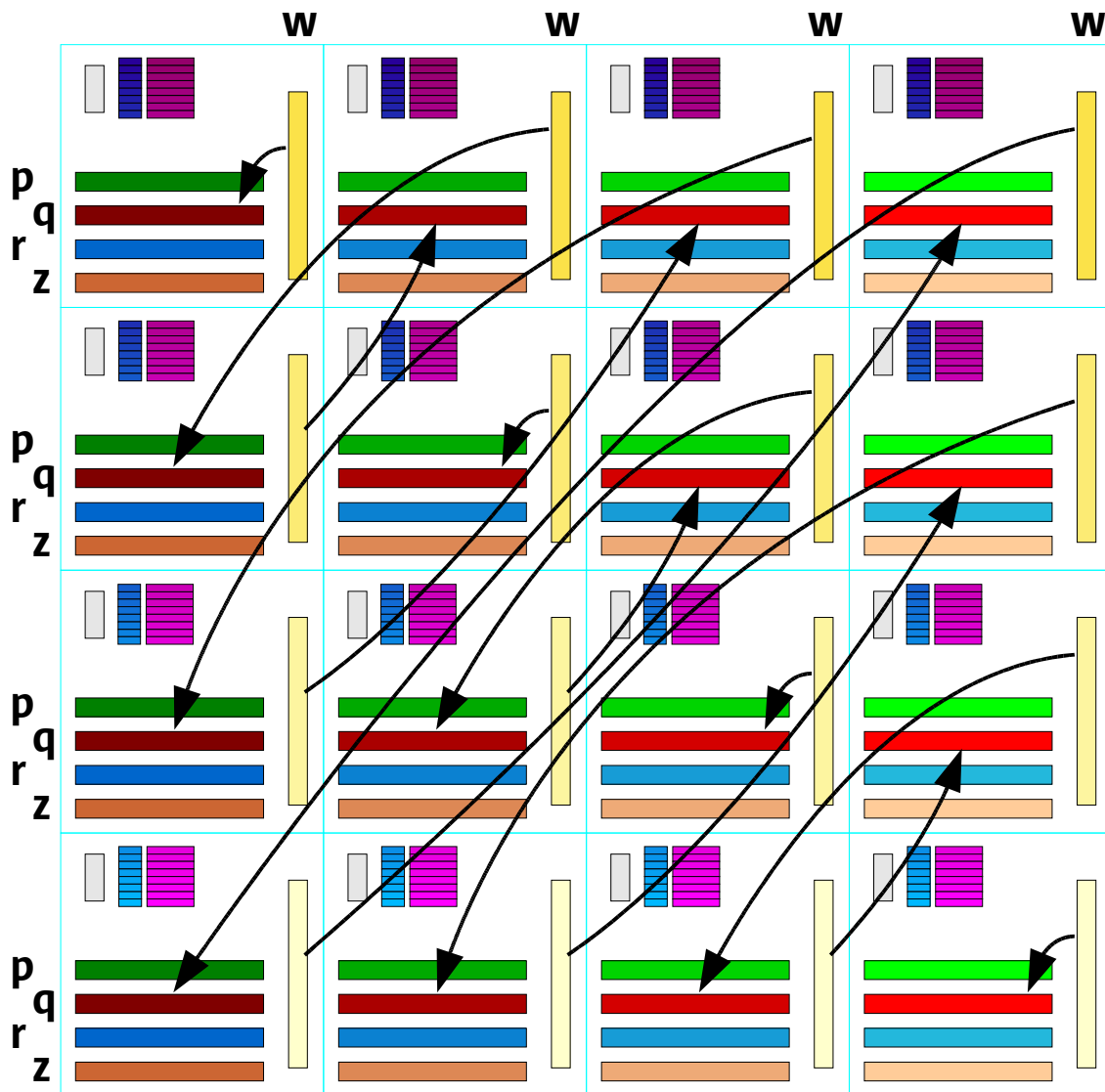
$\| r \| = \| x - A z \|$



# Execution: Matrix-Vector Product (4 of 4)

$z = 0$   
 $r = x$   
 $\rho = r^T r$   
 $p = r$   
**DO**  $i = 1, 25$   
      $q = A p$   
      $\alpha = \rho / (p^T q)$   
      $z = z + \alpha p$   
      $\rho_0 = \rho$   
      $r = r - \alpha q$   
      $\rho = r^T r$   
      $\beta = \rho / \rho_0$   
      $p = r + \beta p$   
**ENDDO**  
 $\| r \| = \| x - A z \|$

transpose  
result into q



# Communication Is Sparse and Regular

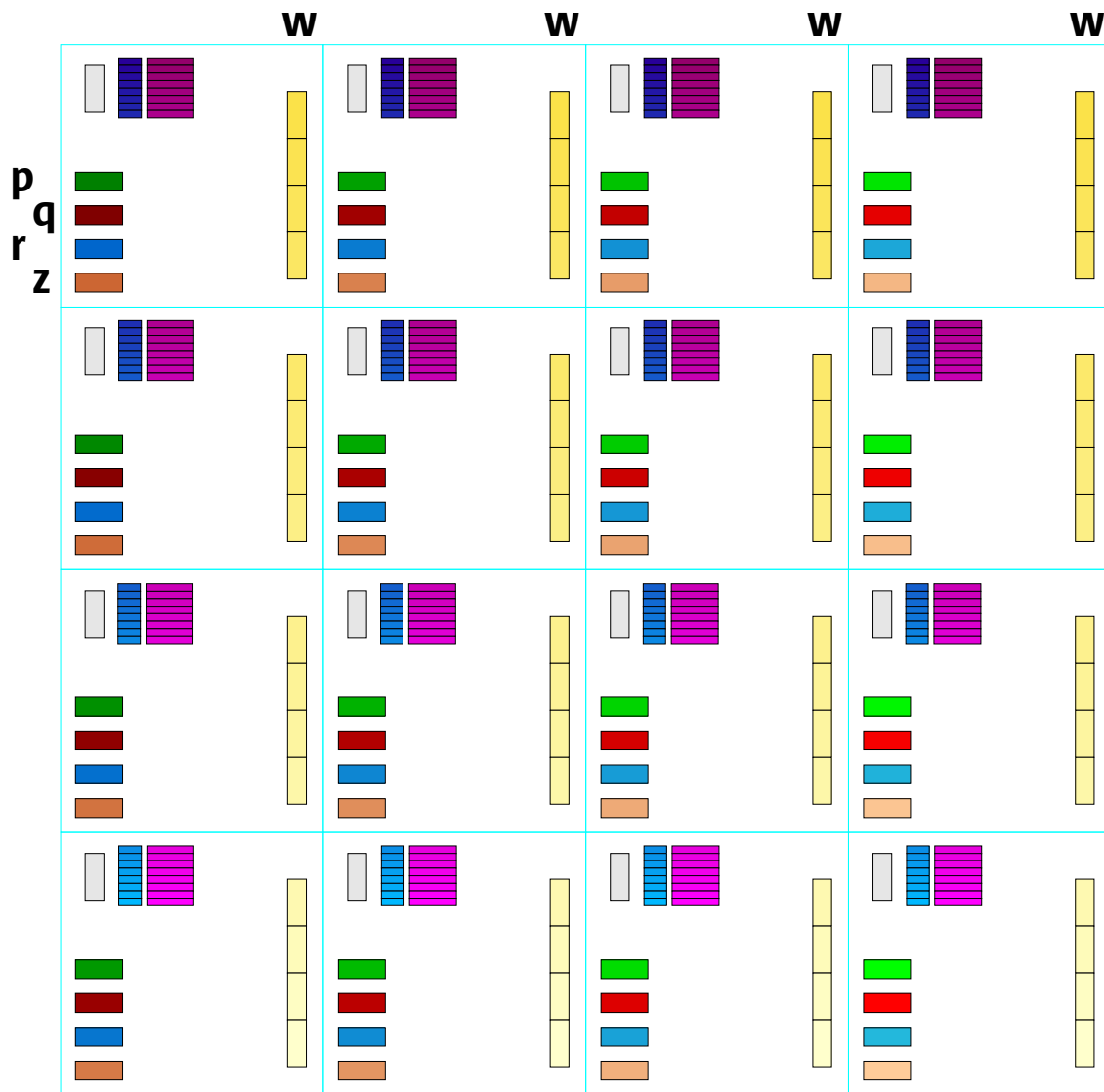
- Each MPI process communicates with only  $\log(P)/2 + 1$  peers
  - >  $\log(P)/2$  neighbors in the same row of the grid, at distances  $\sqrt{P}/2, \sqrt{P}/2 - 1, \dots, 8, 4, 2, 1$
  - > One “transpose buddy”
  - > For  $2^{18}$  processes, that's still just 10 peers
- Communication pattern is regular even though the sparse matrix is random
  - > The randomness of the access pattern is confined to indexing of the vector  $p$ , local to each MPI process
  - > Differs from a benchmark like GUPS, which performs random accesses throughout the global address space

# Duplicated Work in the NPB2.3 MPI Implementation

- Vectors are replicated (as we have seen)
- All dot product and daxpy operations are performed redundantly by a factor of  $\sqrt{P}$
- For large numbers of processes, the redundant work is no longer negligible
  - > Matrix-vector product is about  $\text{NONZER}^2 \cdot \text{NA} / P$  FMAs
  - > Dot-product/daxpy cost is  $5 \cdot \text{NA} / \sqrt{P}$  FMAs
  - > Ratio is  $\text{NONZER}^2 / (5 \cdot \sqrt{P})$
  - > For  $\text{NONZER}=21$  and  $P=64$ , ratio is about **12**
  - > For  $\text{NONZER}=21$  and  $P=2^{18}$ , ratio is about **0.18**

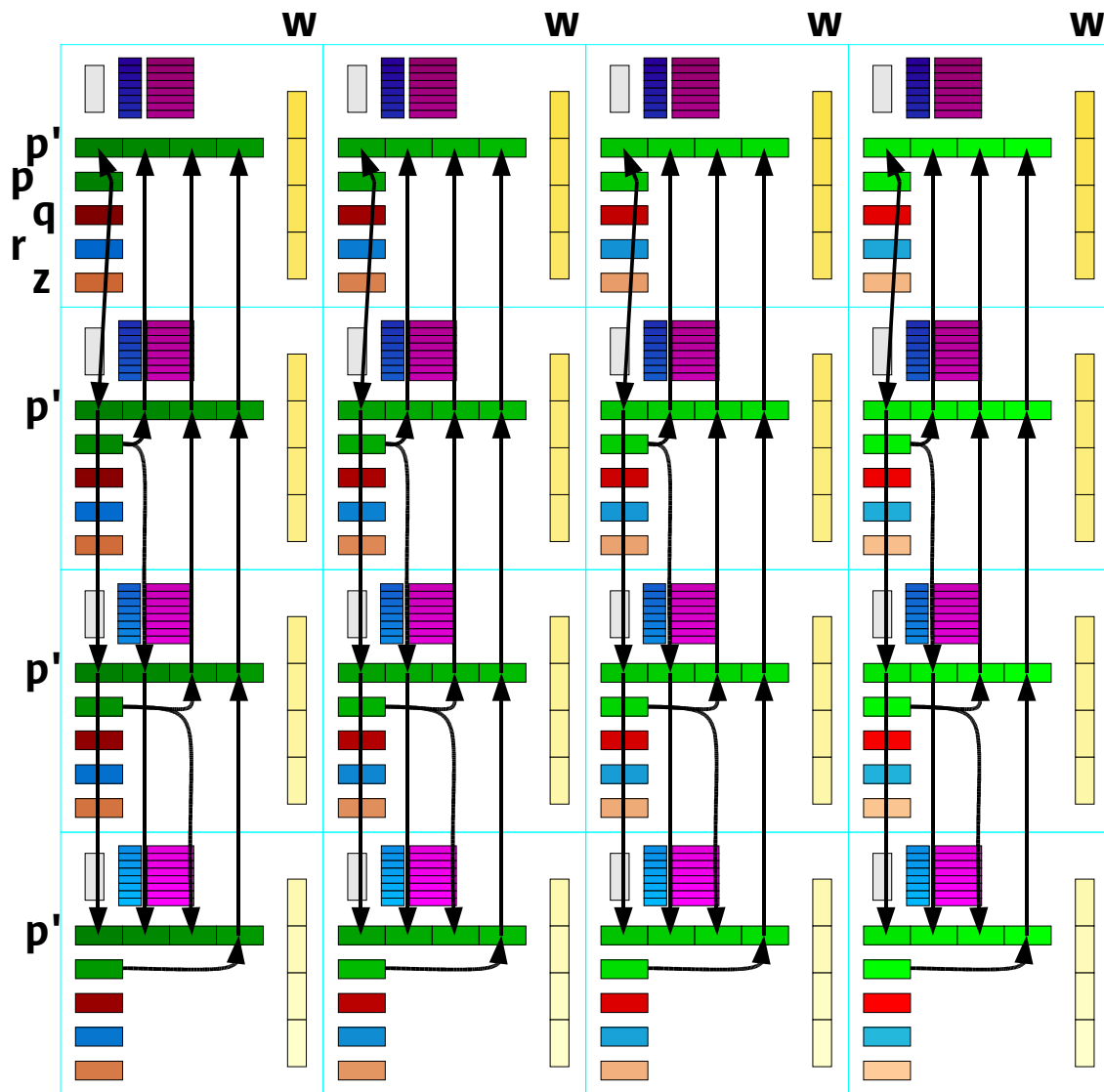
# Alternate MPI Strategy Scales Better

Don't replicate **p**, **q**, **r**, and **z**.  
 Stripe them over all the MPI processes and align them with matrix **A** in column-major order.



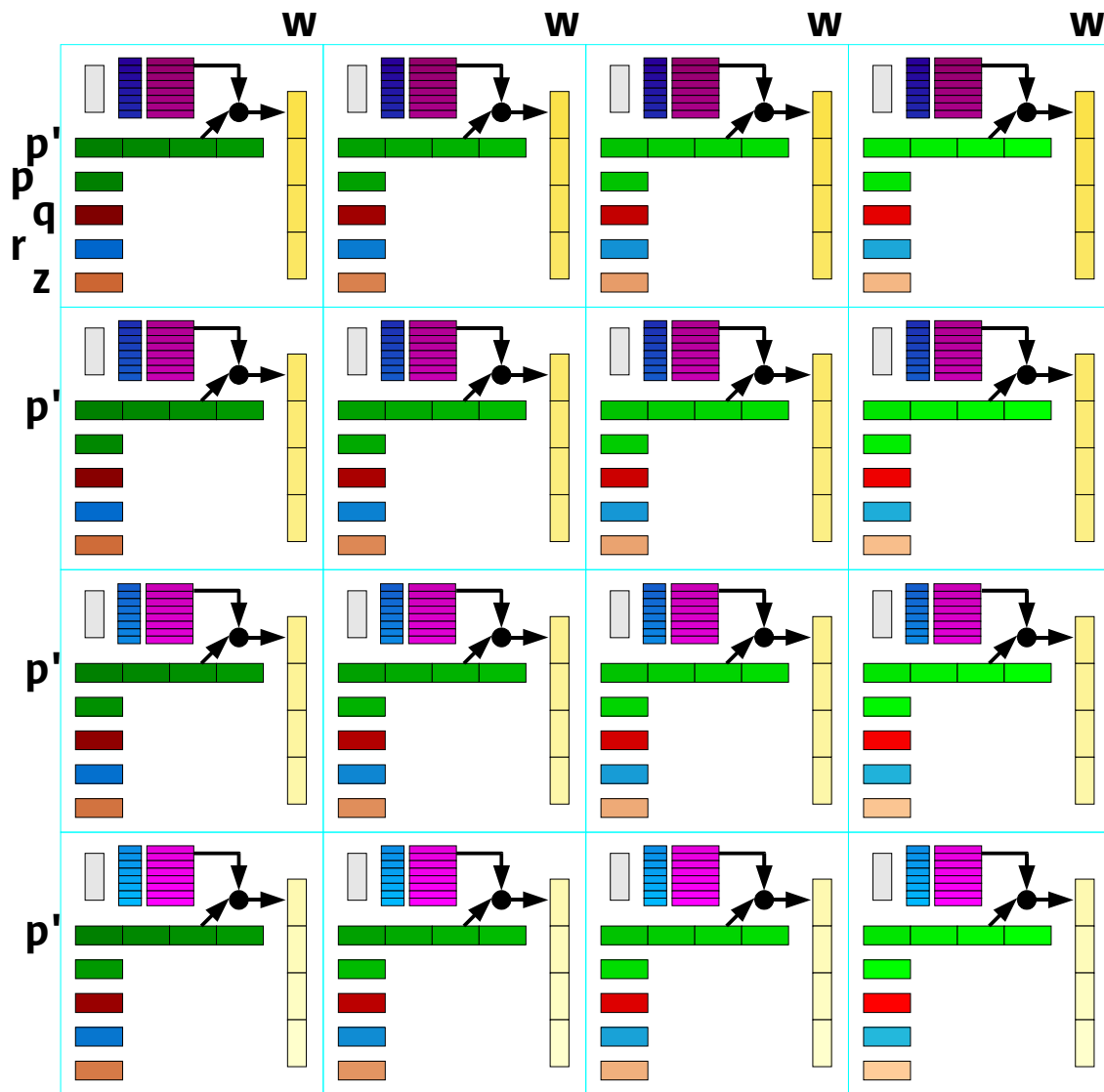
# Alternate Matrix-Vector Multiply (1 of 5)

Perform an MPI\_ALLGATHER operation within each column to replicate  $p$ .



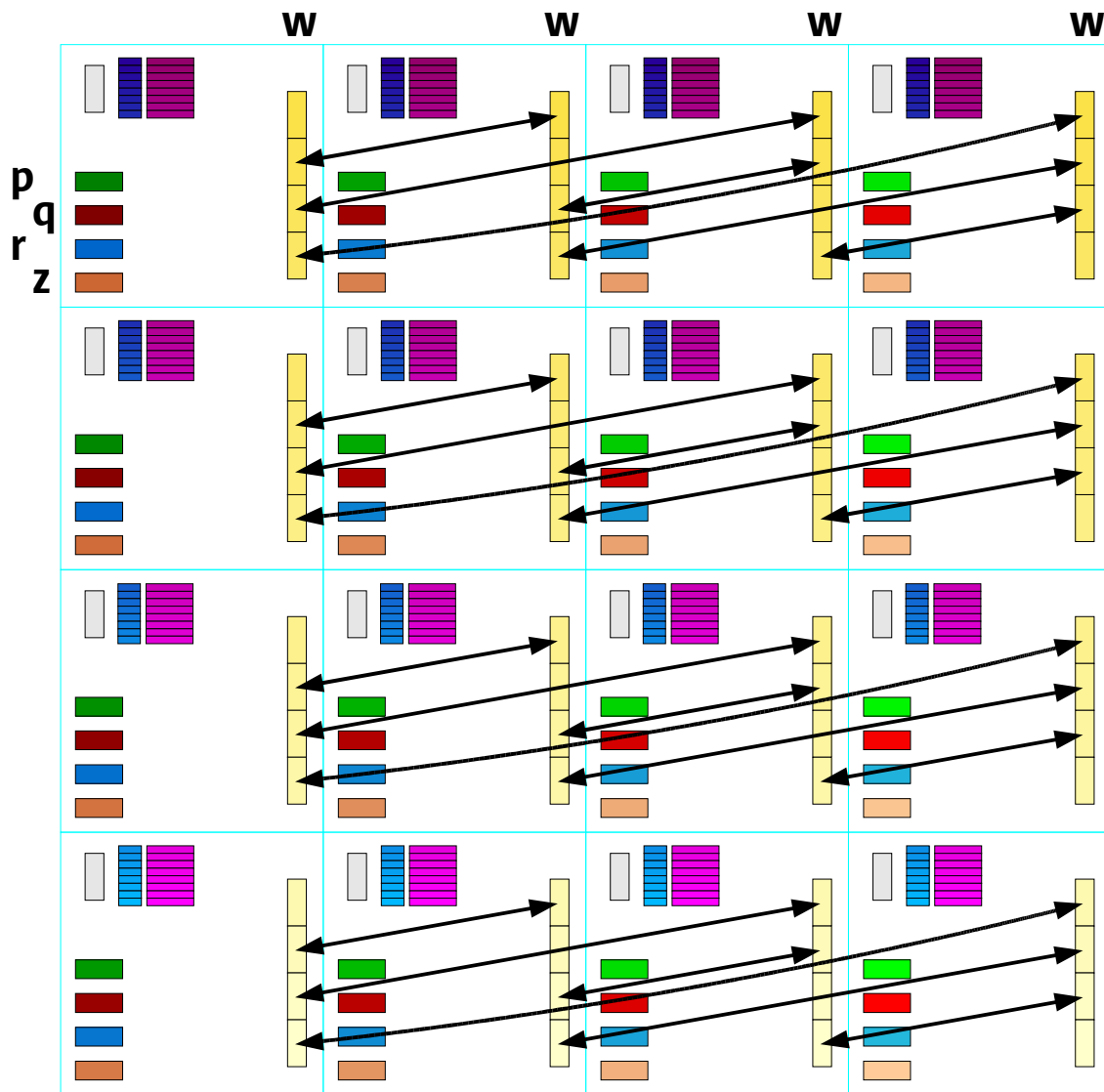
# Alternate Matrix-Vector Multiply (2 of 5)

Compute matrix-vector product local to each tile, yielding  $w$ .



# Alternate Matrix-Vector Multiply (3 of 5)

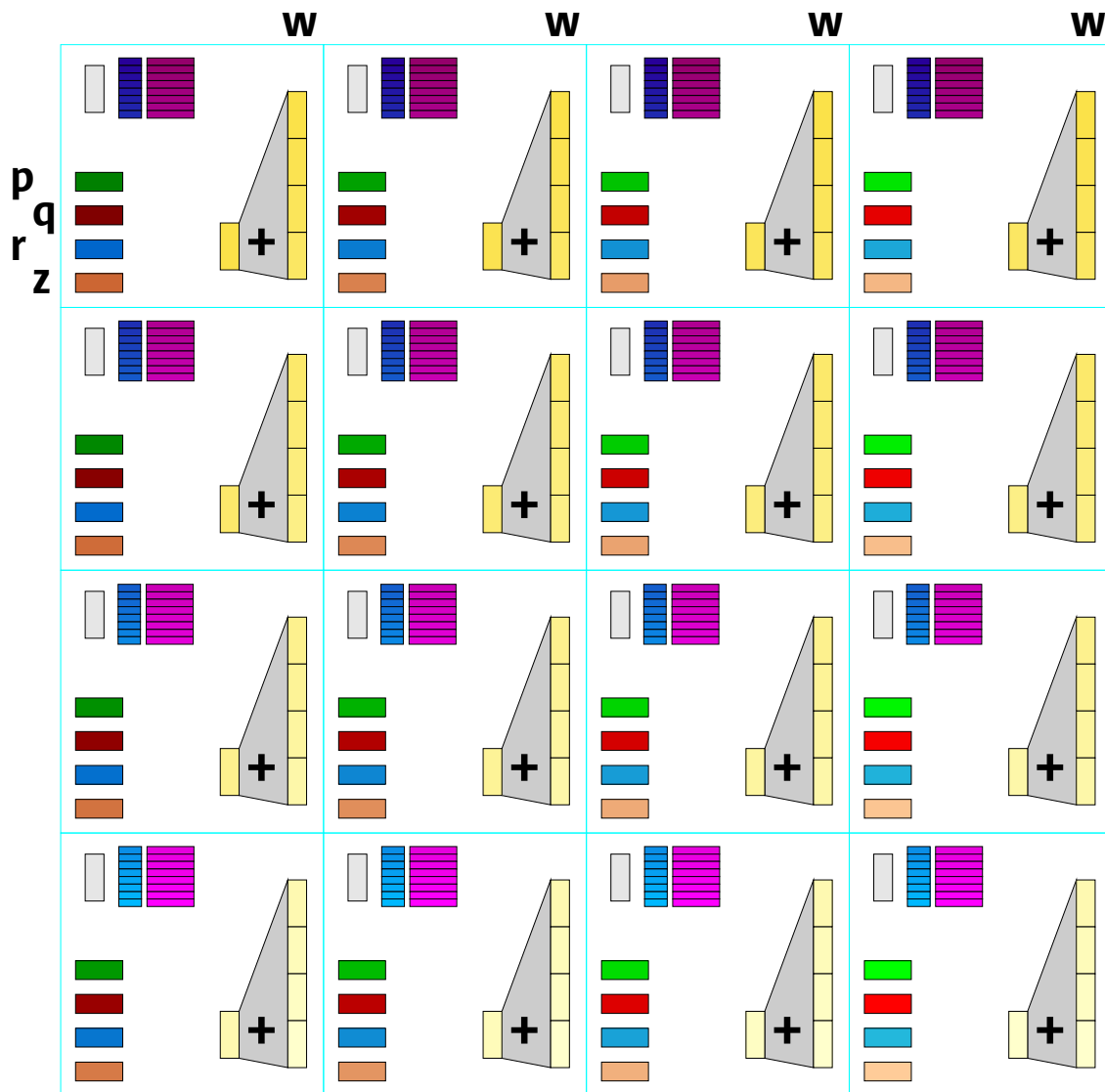
Perform an MPI\_ALLTOALL operation within each row to transpose chunks of **w**.



# Alternate Matrix-Vector Multiply (4 of 5)

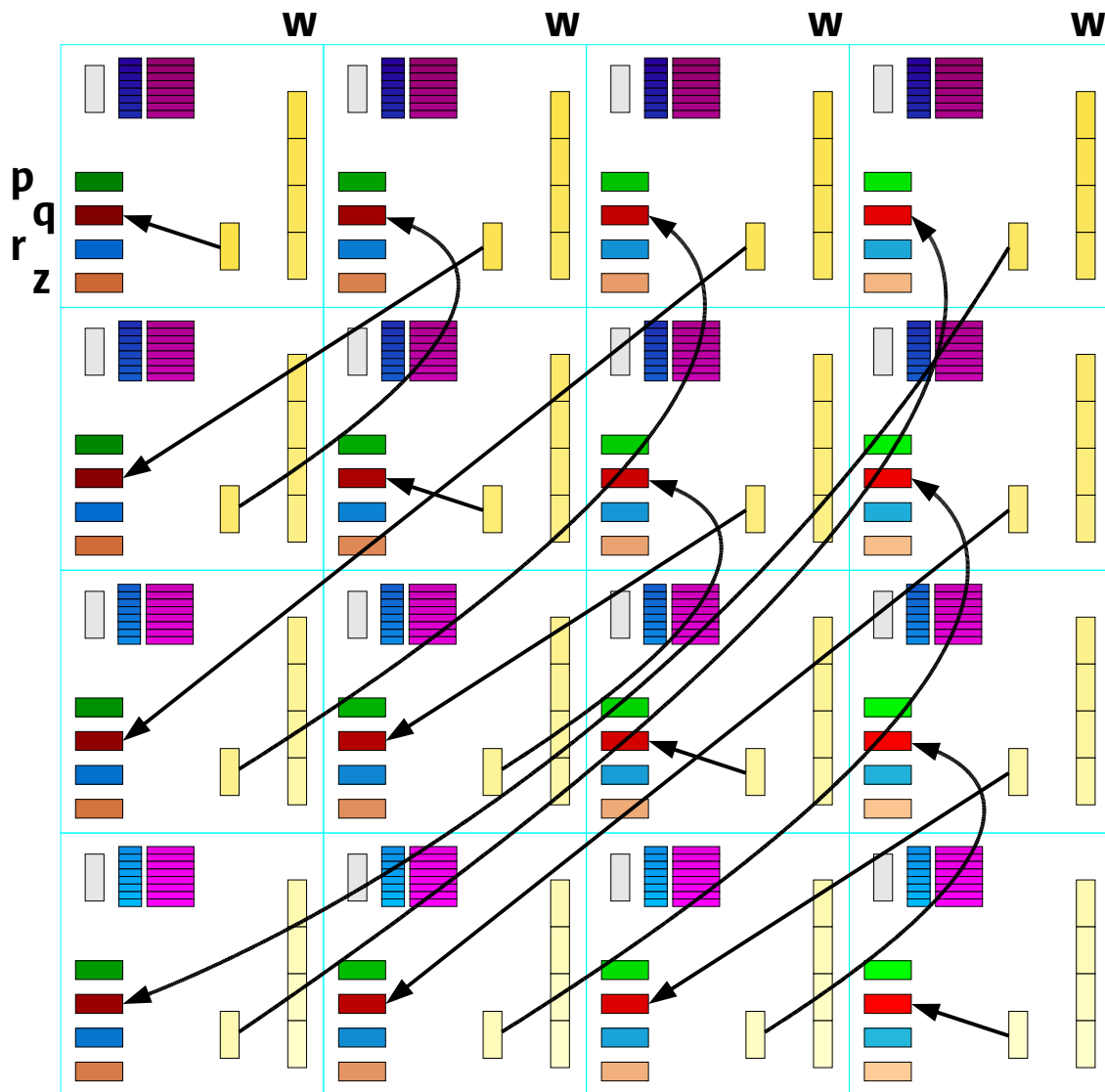
Elementwise  
sum chunks of  
 $w$  locally within  
each tile.

(Actually, the  
preceding  
MPI\_ALLTOALL  
step and this  
step can be  
done as a single  
MPI\_REDUCE\_  
SCATTER step.)



# Alternate Matrix-Vector Multiply (5 of 5)

Point-to-point  
send transposes  
sums and stores  
them into **q**.



# Communication Differences

- Alternate method communicates with  $2\sqrt{P}-1$  peers rather than  $\log(P)/2+1$ 
  - > Still fairly sparse if  $P$  large
- But vectors communicated have length  $N/P$  rather than  $N/\sqrt{P}$ 
  - > Potential data-transfer win of  $O(\log P)$
- Tradeoffs remain
  - > Matrix-vector product needs fewer synch steps
  - > Dot product requires twice as many steps
  - > Which method is faster overall depends on both problem parameters and machine parameters

# Question

Can we capture this kind of knowledge  
in libraries?

# Our Vision

- CG application code written in simple mathematical style
- Library provides data types “vector” and “sparse matrix”
- Construction of sparse matrix automatically distributes it for parallel processing
- Library provides tuned parallel algorithms for matrix-vector product and dot product
- Compiler, RTE, profiler, library (and programmer?) cooperate to compute optimized data layouts

# Matrix Multiply Makes Choices

- Matrix multiplication dynamically chooses among a smallish number of algorithms
  - > Based on pre-analysis of matrix
  - > Based on representation and layout
  - > Based on hardware resources
- Overhead of dynamic choices is minimized by dynamic recompilation of choice trees
  - > Standard technology in some Java<sup>TM</sup> Virtual Machines today

# Matrix Multiply Uses Library, Too

- Matrix multiplication is not built from scratch, but itself relies on libraries for replication, sum-reduction, and so on
- Well-factored, reuseable code
- One important angle is parameterizing the data type of elements while maintaining performance

# Encapsulation in Tuned Library

- With key algorithms properly factored and encapsulated in tuned libraries, application code is almost entirely unchanged from the “simple code” previously shown
- Big idea is exposing algorithmic decisions in libraries rather than burying them in compilers
- Other big idea is burying algorithmic decisions in libraries rather than exposing them in applications
- We want to start a cult of “application code is simple”

# Parallelism Is Not a Feature!

- Parallel programming is not a goal, but a pragmatic compromise.
- It would be a lot easier to program a single processor chip running at 1 PHz than a million processors running at 20 GHz.
  - > We don't know how to build a 1 Phz processor.
  - > Even if we did, someone would still want to strap a bunch of them together!
- Parallel programming is difficult and error-prone. (This is not a property of machines, but of people.)

# Where to Put the Expertise?

- High Performance Fortran
  - > Fairly concise notation, global point of view
  - > Tried to capture all knowledge in the compiler
  - > Fixed set of distribution declarations (not extensible)
- Fortran + MPI
  - > Everything is possible, but burden is on programmer
  - > Local point of view; hard to abstract parallelism
- Fortress
  - > Maniacally concise notation, global point of view
  - > Tries to capture all knowledge in libraries, not compiler
  - > Compiler provides facilities to library writers, optimizes

# Our Key Design Themes

- Make stupid mistakes impossible
  - And make clever mistakes relatively unlikely
- Design the language to be grown by users
  - Rich library language enables simple application languages
- Make abstraction efficient
  - Aggressive static and dynamic optimization
- Make parallelism tractable
  - Identify and support standard communication patterns / data types
- Emulate standard mathematical notation
  - Reduce the effort of translating from science to computation

[guy.steele@sun.com](mailto:guy.steele@sun.com)

[http://research.sun.com/  
projects/plrg](http://research.sun.com/projects/plrg)