Communication-optimal parallel 2.5D matrix multiplication and LU factorization algorithms

Edgar Solomonik and James Demmel

UC Berkeley

September 1st, 2011

Outline

Solving science problems faster

Parallel computers can solve **bigger** problems

- **weak scaling**

Parallel computers can also solve a fixed problem **faster**

- **strong scaling**

Obstacles to strong scaling

- may increase relative cost of **communication**
- may hurt load balance

Achieving strong scaling

How to reduce communication and maintain load balance?

- reduce communication along the critical path

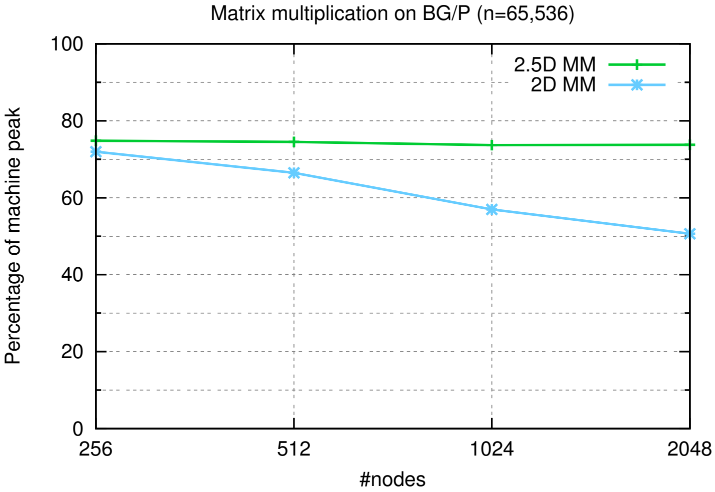
Communicate **less**

- *avoid* unnecessary communication

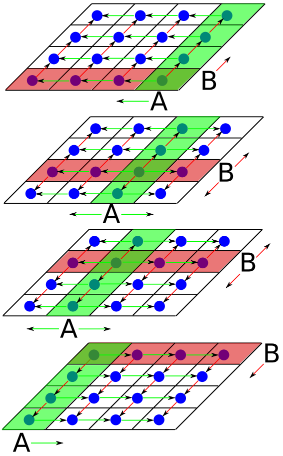
Communicate **smarter**

- know your network topology

Strong scaling matrix multiplication

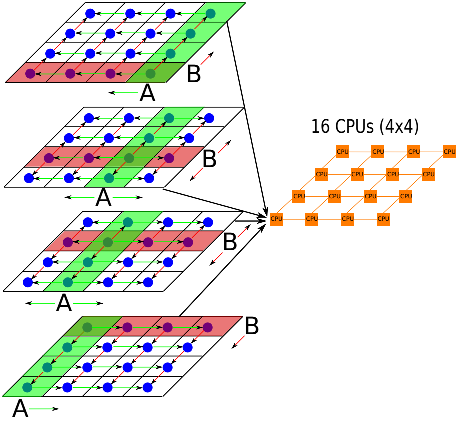


Blocking matrix multiplication



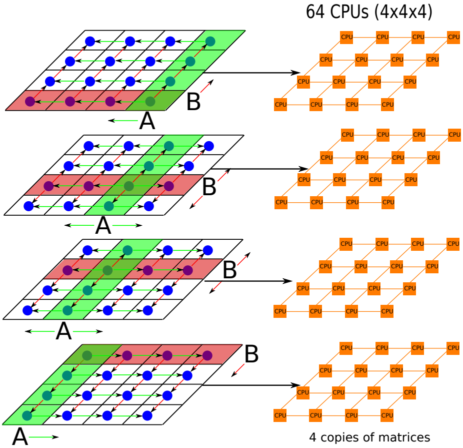
2D matrix multiplication

[Cannon 69], [Van De Geijn and Watts 97]

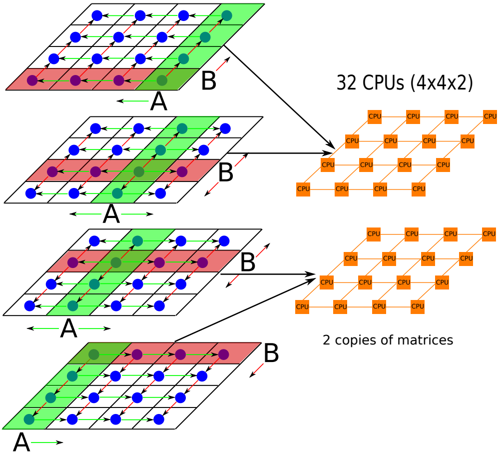


3D matrix multiplication

[Agarwal et al 95], [Aggarwal, Chandra, and Snir 90], [Bernsten 89]



2.5D matrix multiplication



2.5D strong scaling

n = dimension, p = #processors, c = #copies of data

- must satisfy $1 \le c \le p^{1/3}$
- special case: $c = 1$ yields 2D algorithm
- special case: $c = p^{1/3}$ yields 3D algorithm

$$\begin{aligned} \text{cost}(\text{2.5D MM}(p, c)) = \ & O(n^3/p) \text{ flops} \\ & + O(n^2/\sqrt{c \cdot p}) \text{ words moved} \\ & + O(\sqrt{p/c^3}) \text{ messages}^* \end{aligned}$$

*ignoring log(p) factors

2.5D strong scaling

n = dimension, p = #processors, c = #copies of data

- must satisfy $1 \leq c \leq p^{1/3}$
- special case: $c = 1$ yields 2D algorithm
- special case: $c = p^{1/3}$ yields 3D algorithm

$$\begin{aligned}\text{cost}(\text{2D MM}(p)) &= O(n^3/p) \text{ flops} \\ &\quad + O(n^2/\sqrt{p}) \text{ words moved} \\ &\quad + O(\sqrt{p}) \text{ messages}^* \\ &= \text{cost}(\text{2.5D MM}(p, 1))\end{aligned}$$

*ignoring log(p) factors

2.5D strong scaling

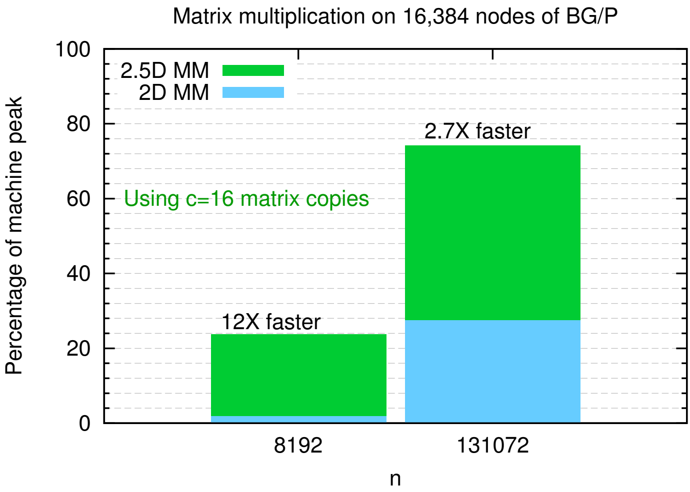
n = dimension, p = #processors, c = #copies of data

- must satisfy $1 \le c \le p^{1/3}$
- special case: $c = 1$ yields 2D algorithm
- special case: $c = p^{1/3}$ yields 3D algorithm

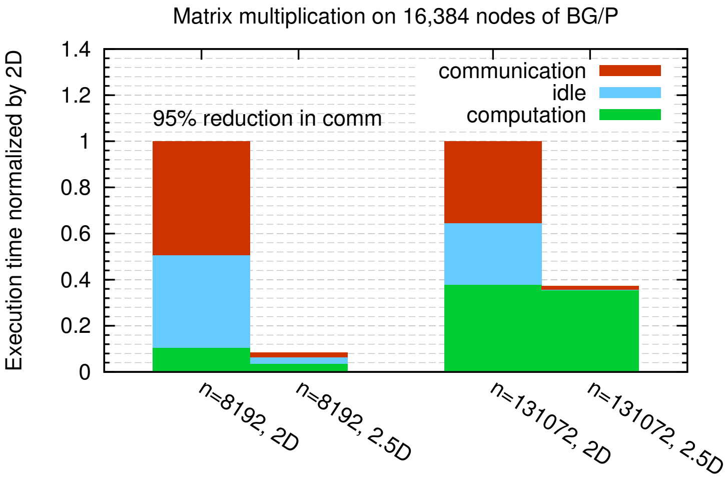
$$\begin{aligned}\text{cost}(\text{2.5D MM}(\mathbf{c} \cdot p, \mathbf{c})) &= O(n^3/(\mathbf{c} \cdot p)) \text{ flops} \\ &\quad + O(n^2/(\mathbf{c} \cdot \sqrt{p})) \text{ words moved} \\ &\quad + O(\sqrt{p}/\mathbf{c}) \text{ messages} \\ &= \text{cost}(\text{2D MM}(p))/\mathbf{c}\end{aligned}$$

perfect strong scaling

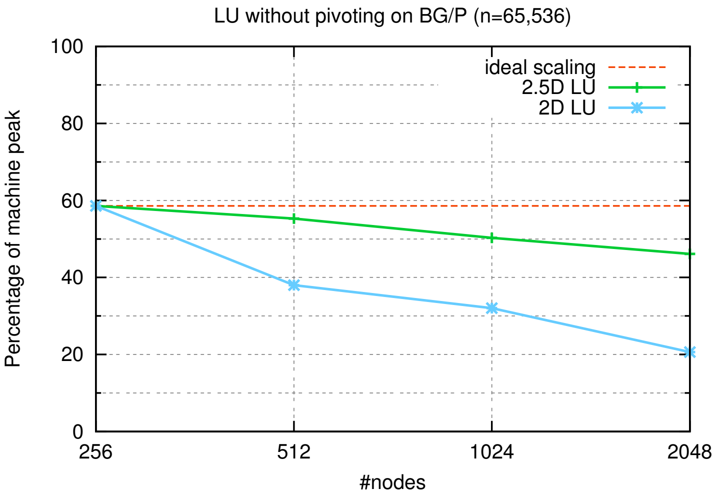
2.5D MM on 65,536 cores



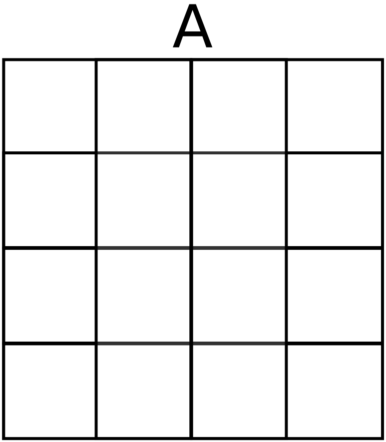
Cost breakdown of MM on 65,536 cores



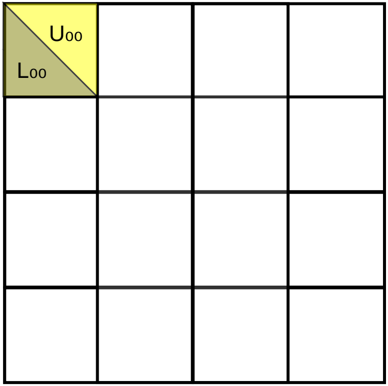
2.5D LU strong scaling (without pivoting)



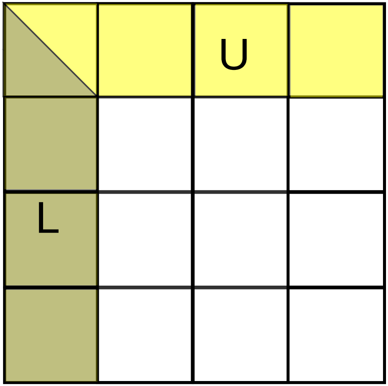
2D blocked LU factorization



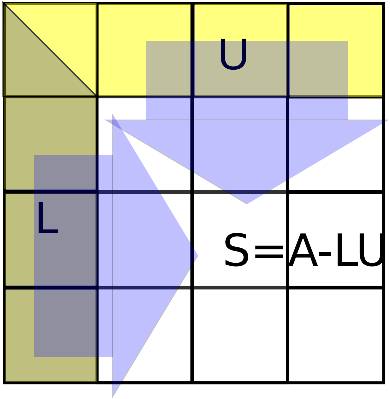
2D blocked LU factorization



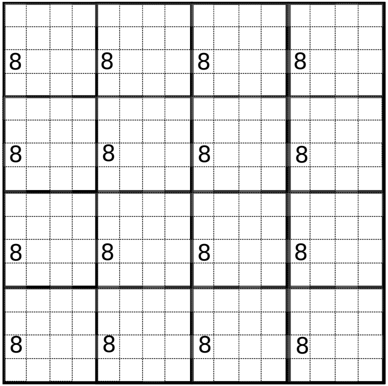
2D blocked LU factorization



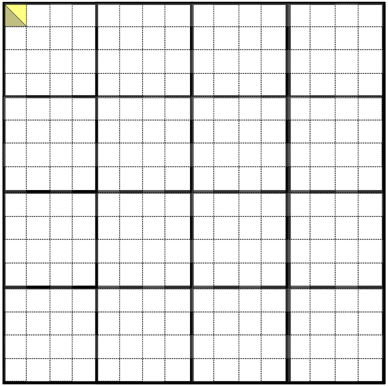
2D blocked LU factorization



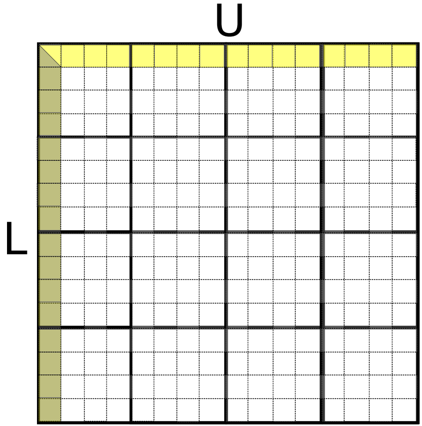
2D block-cyclic decomposition



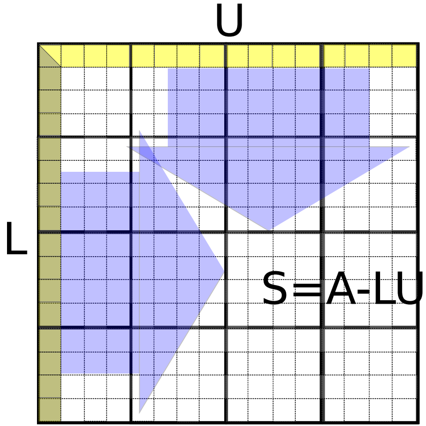
2D block-cyclic LU factorization



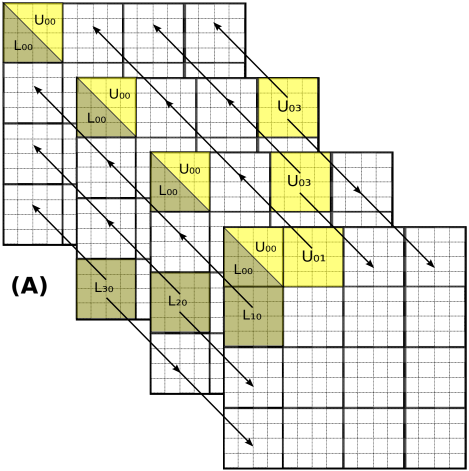
2D block-cyclic LU factorization



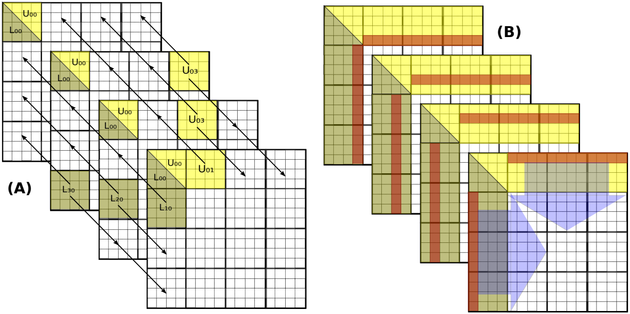
2D block-cyclic LU factorization



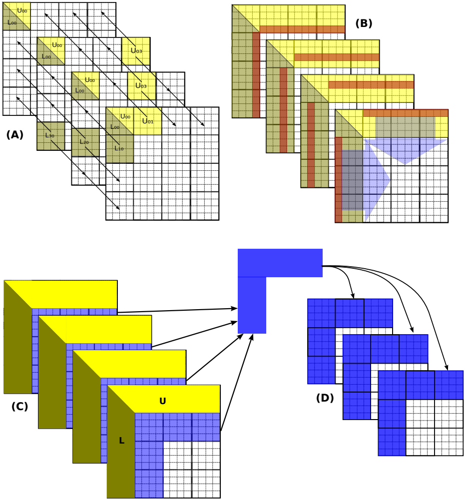
2.5D LU factorization



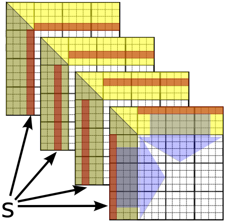
2.5D LU factorization



2.5D LU factorization



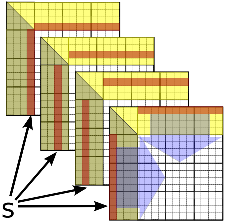
2.5D LU factorization



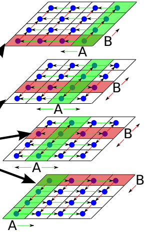
Look at how this
update is distributed.

What does it remind you of?

2.5D LU factorization



Look at how this update is distributed.



Same 3D update in multiplication

Communication-avoiding pivoting

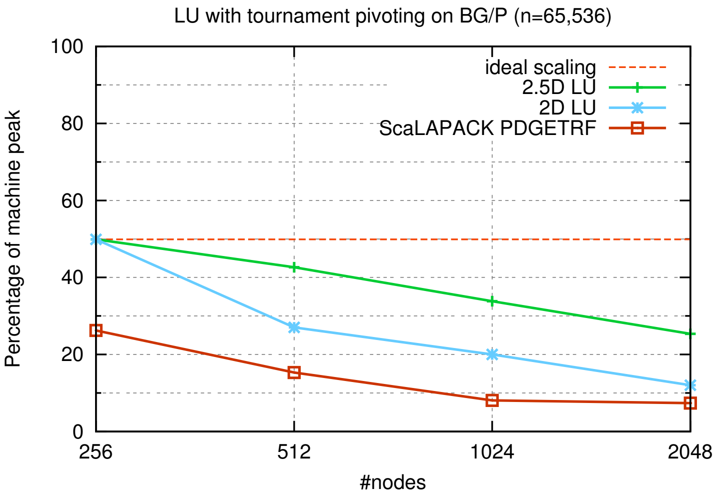
Partial pivoting is not communication-optimal on a blocked matrix

- require message/synchronization for each column
- $O(n)$ messages required

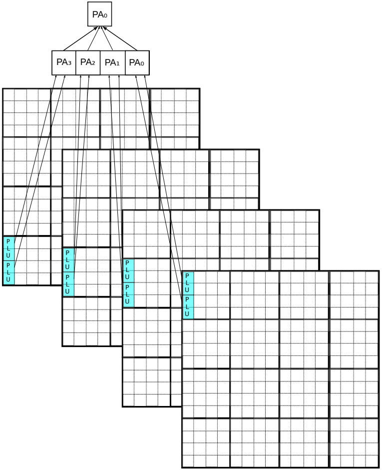
Tournament pivoting or Communication-Avoiding (CA) pivoting

- performs a tournament to determine best pivot row candidates
- blocked CA-pivoting algorithm is communication-optimal

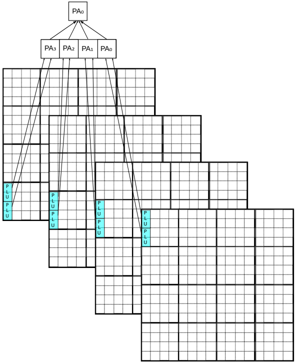
Strong scaling of 2.5D LU with tournament pivoting

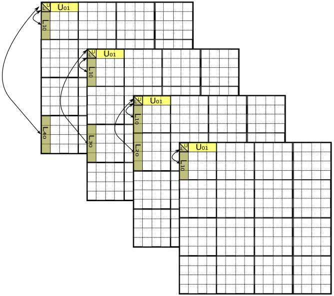


2.5D LU factorization with tournament pivoting

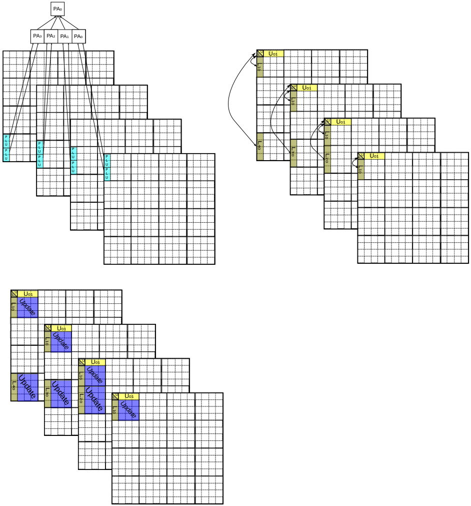


2.5D LU factorization with tournament pivoting

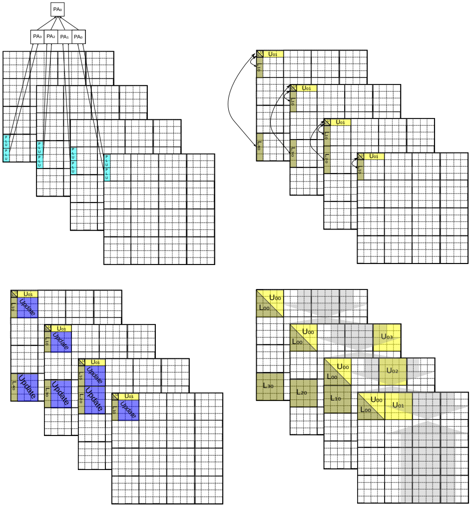




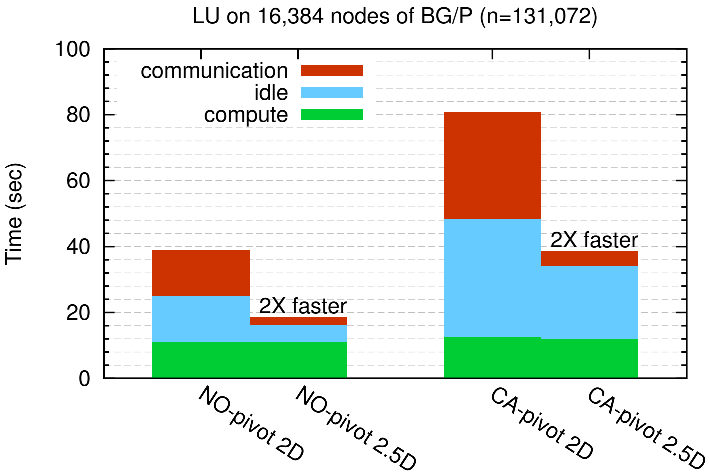
2.5D LU factorization with tournament pivoting



2.5D LU factorization with tournament pivoting



2.5D LU on 65,536 cores



Conclusion

Our contributions:

- 2.5D mapping of matrix multiplication
  - Optimal according to lower bounds in [Irony, Tiskin, Toledo 04] and [Aggarwal, Chandra, and Snir 90]
- A new latency lower bound for LU
- Communication-optimal 2.5D LU
  - Bandwidth-optimal according to general lower bound [Ballard, Demmel, Holtz, Schwartz 10]
  - Latency-optimal according to new lower bound

Open questions:

- 2.5D Householder QR

Reflections:

- Replication allows better strong scaling
- Topology-aware mapping cuts communication costs

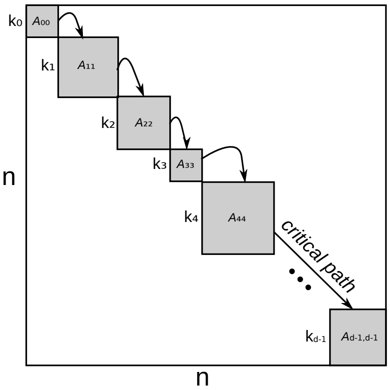
Backup slides

A new latency lower bound for LU

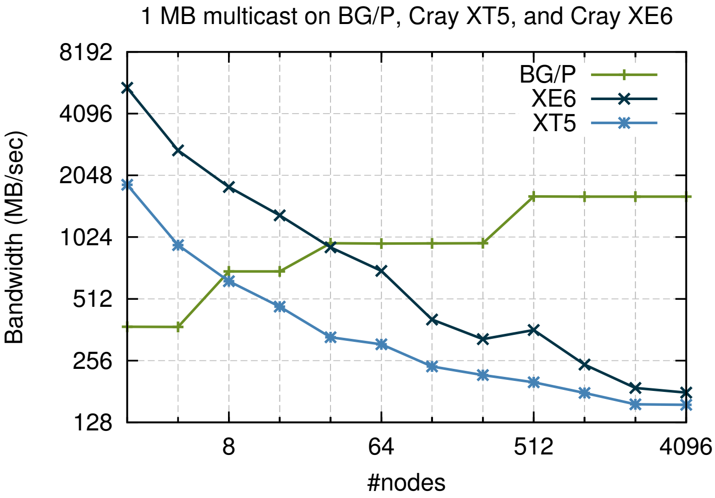
LU with $O(\sqrt{P/c^3})$ messages?

- For block size $n/\mathbf{d}$ LU does
  - $\Omega(n^3/\mathbf{d}^2)$ flops
  - $\Omega(n^2/\mathbf{d})$ words
  - $\Omega(\mathbf{d})$ msgs
- Now pick $\mathbf{d}$ (=latency cost)
  - $\mathbf{d} = \boldsymbol{\Omega}(\sqrt{\mathbf{P}})$ to minimize flops
  - $\mathbf{d} = \boldsymbol{\Omega}(\sqrt{\mathbf{c} \cdot \mathbf{P}})$ to minimize words

No dice. Lets minimize bandwidth.



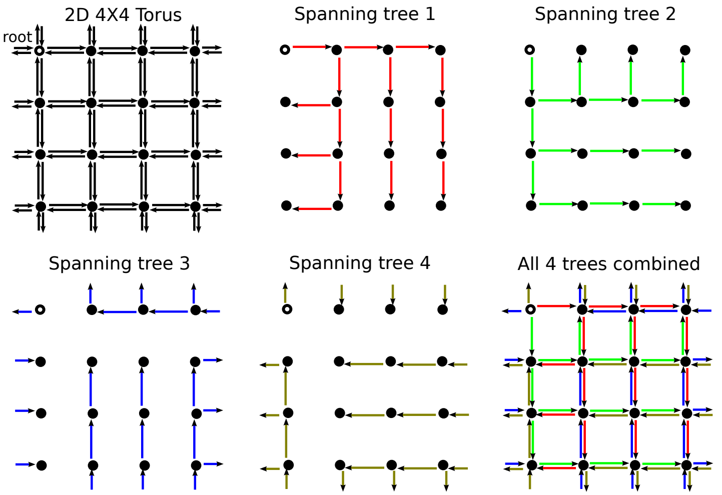
Performance of multicast (BG/P vs Cray)



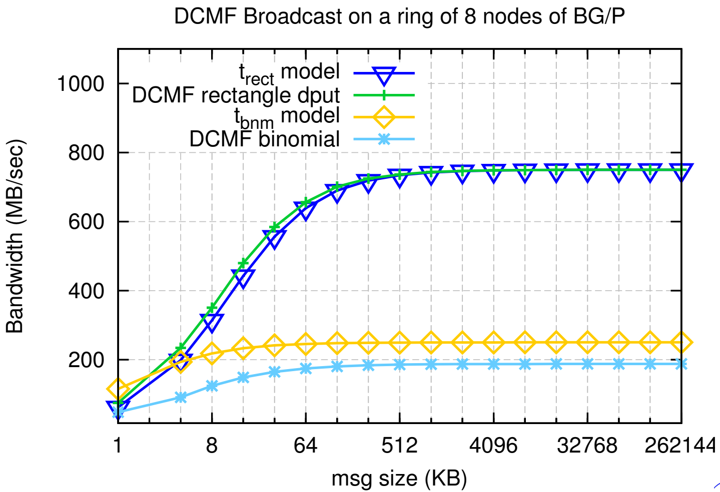
Why the performance discrepancy in multicasts?

- Cray machines use **binomial multicasts**
  - Form spanning tree from a list of nodes
  - Route copies of message down each branch
  - Network contention degrades utilization on a 3D torus
- BG/P uses **rectangular multicasts**
  - Require network topology to be a k-ary n-cube
  - Form $2n$ edge-disjoint spanning trees
    - Route in different dimensional order
    - Use both directions of bidirectional network

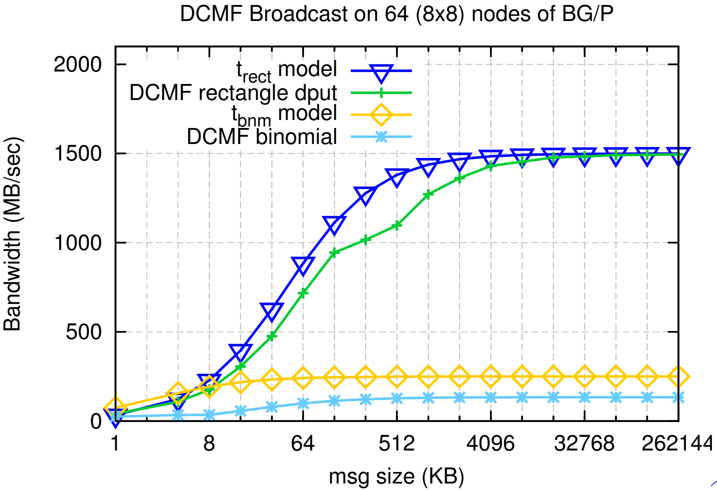
2D rectangular multicasts trees



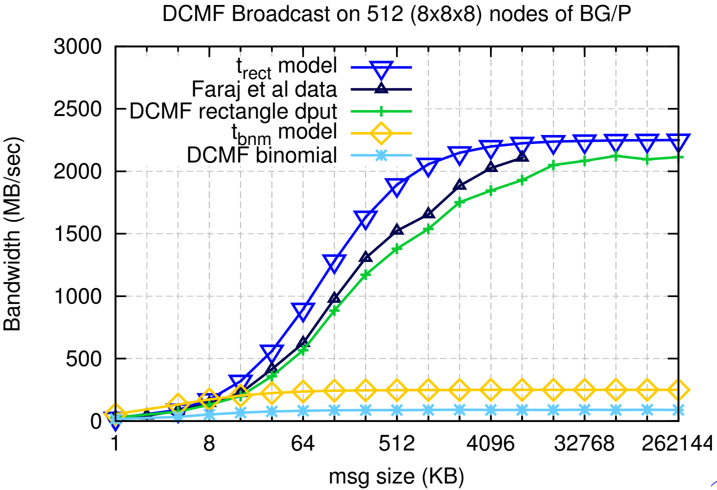
Model verification: one dimension



Model verification: two dimensions



Model verification: three dimensions

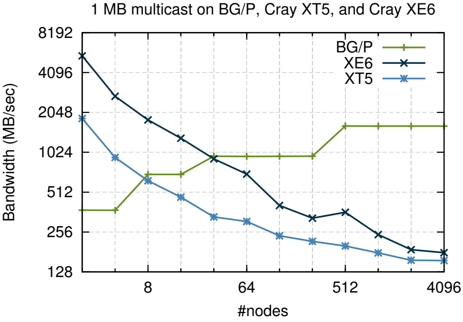


Another look at that first plot

Just how much better are rectangular algorithms on $P = 4096$ nodes?

- Binomial collectives on XE6
  - **1/30th of link bandwidth**
- Rectangular collectives on BG/P
  - **4.3X the link bandwidth**
- **Over 120X improvement in efficiency!**

How can we apply this?



Bridging dense linear algebra techniques and applications

Target application: tensor contractions in electronic structure calculations (quantum chemistry)

- Often memory constrained
- Most target tensors are oddly shaped
- Need support for high dimensional tensors
- Need handling of partial/full tensor symmetries
- Would like to use communication avoiding ideas (blocking, 2.5D, topology-awareness)

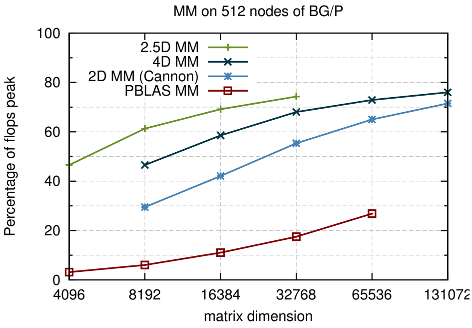
Decoupling memory usage and topology-awareness

- 2.5D algorithms couple memory usage and virtual topology
  - c copies of a matrix implies c processor layers
- Instead, we can nest 2D and/or 2.5D algorithms
- Higher-dimensional algorithms allow smarter topology aware mapping

4D SUMMA-Cannon

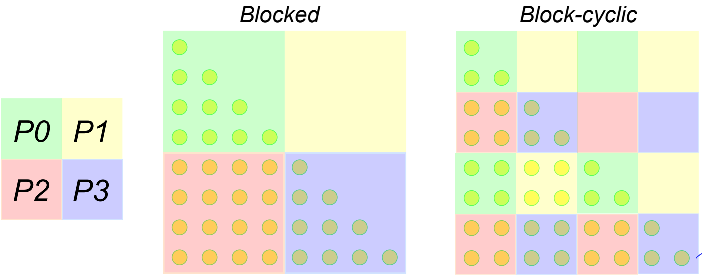
How do we map to a 3D partition *without using more memory*

- SUMMA (bcast-based) on 2D layers
- Cannon (send-based) along third dimension
- Cannon calls SUMMA as sub-routine
  - Minimize inefficient (non-rectangular) communication
  - Allow better overlap
- Treats MM as a 4D tensor contraction



Symmetry is a problem

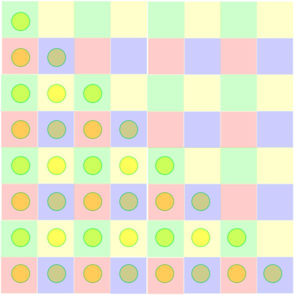
- A fully symmetric tensor of dimenson d requires only $n^d/d!$ storage
- Symmetry significantly complicates sequential implementation
  - Irregular indexing makes alignment and unrolling difficult
  - Generalizing over all partial-symmetries is expensive
- Blocked or block-cyclic virtual processor decmpositions give irregular or imbalanced virtual grids



Solving the symmetry problem

- A **cyclic decomposition** allows balanced and regular blocking of symmetric tensors
- If the cyclic-phase is the same in each symmetric dimension, each sub-tensor retains the symmetry of the whole tensor

Cyclic



A generalized cyclic layout is still challenging

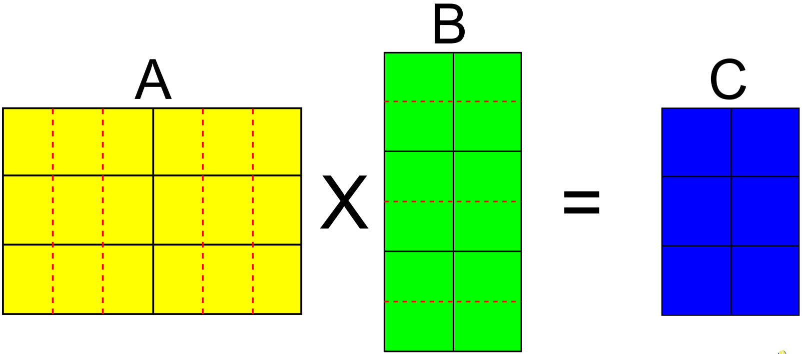
- In order to retain partial symmetry, all symmetric dimensions of a tensor must be mapped with the same cyclic phase
- The contracted dimensions of A and B must be mapped with the same phase
- And yet the virtual mapping, needs to be mapped to a physical topology, which can be any shape

Virtual processor grid dimensions

- Our virtual cyclic topology is somewhat restrictive and the physical topology is very restricted
- Virtual processor grid dimensions serve as a new level of indirection
  - If a tensor dimension must have a certain cyclic phase, adjust physical mapping by creating a virtual processor dimension
  - Allows physical processor grid to be 'stretchable'

Constructing a virtual processor grid for MM

Matrix multiply on 2x3 processor grid. Red lines represent virtualized part of processor grid. Elements assigned to blocks by cyclic phase.



Unfolding the processor grid

- Higher-dimensional fully-symmetric tensors can be mapped onto a lower-dimensional processor grid via creation of new virtual dimensions
- Lower-dimensional tensors can be mapped onto a higher-dimensional processor grid via by unfolding (serializing) pairs of processor dimensions
- However, when possible, replication is better than unfolding, since unfolded processor grids can lead to an unbalanced mapping

A basic parallel algorithm for symmetric tensor contractions

1. Arrange processor grid in any k-ary n-cube shape
2. Map (via unfold & virt) both A and B cyclically along the dimensions being contracted
3. Map (via unfold & virt) the remaining dimensions of A and B cyclically
4. For each tensor dimension contracted over, recursively mulitply the tensors along the mapping
    - Each contraction dimension is represented with a nested call to a local multiply or a parallel algorithm (e.g. Cannon)

Tensor library structure

The library supports arbitrary-dimensional parallel tensor contractions with any symmetries on n-cuboid processor torus partitions

1. Load tensor data by (global rank, value) pairs
2. Once a contraction is defined, map participating tensors
3. Distribute or reshuffle tensor data/pairs
4. Construct contraction algorithm with recursive function/args pointers
5. Contract the sub-tensors with a user-defined sequential contract function
6. Output (global rank, value) pairs on request

Current tensor library status

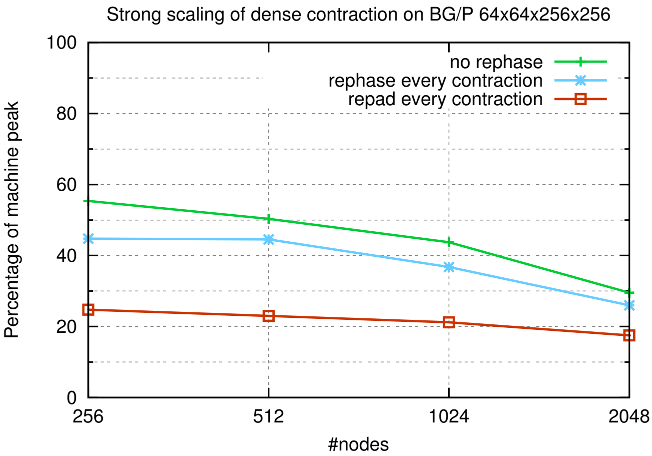
- Dense and symmetric remapping/repadding/contractions implemented
- Currently functional only for dense tensors, but with full symmetric logic
- Can perform automatic mapping with physical and virtual dimensions, but cannot unfold processor dimensions yet
- Complete library interface implemented, including basic auxillary functions (e.g. map/reduce, sum, etc.)

Next implementation steps

- Currently integrating library with a SCF method code that uses dense contractions
- Get symmetric redistribution working correctly
- Automatic unfolding of processor dimensions
- Implement mapping by replication to enable 2.5D algorithms
- Much basic performance debugging/optimization left to do
- More optimization needed for sequential symmetric contractions

Very preliminary contraction library results

Contracts tensors of size $64x64x256x256$ in 1 second on 2K nodes



Potential benefit of unfolding

Unfolding smallest two BG/P torus dimensions improves performance.