Reducing communication in dense matrix/tensor computations

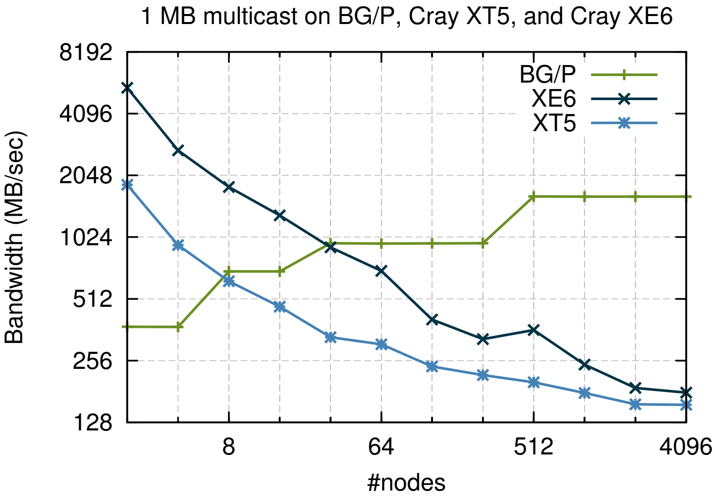
Edgar Solomonik

UC Berkeley

Aug 11th, 2011

Outline

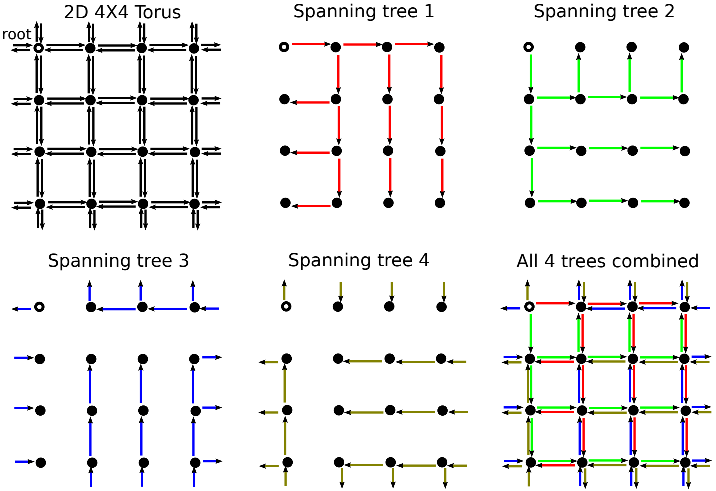
Performance of multicast (BG/P vs Cray)



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



A model for rectangular multicasts

$$t_{mcast} = m/B_n + 2(d+1)\cdot o + 3L + d\cdot P^{1/d}\cdot(2o+L)$$

Our multicast model consists of 3 terms

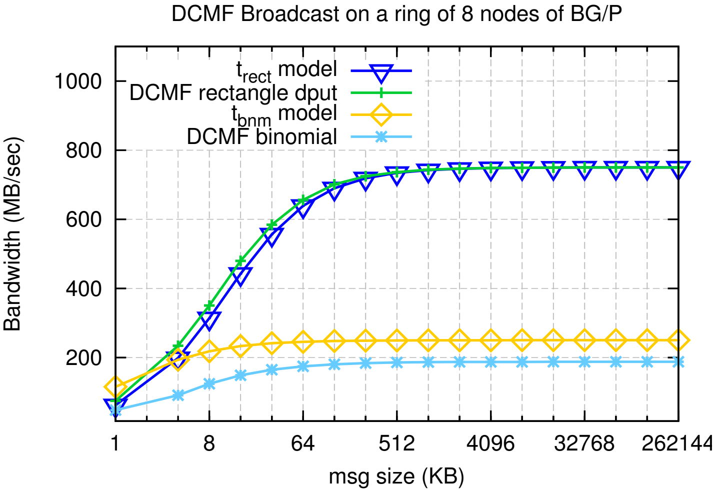
1. m/B_n, the bandwidth cost incurred at the root
2. $2(d+1)\cdot o + 3L$, the start-up overhead of setting up the multicasts in all dimensions
3. $d\cdot P^{1/d}\cdot(2o+L)$, the path overhead reflects the time for a packet to get from the root to the farthest destination node

A model for binomial multicasts

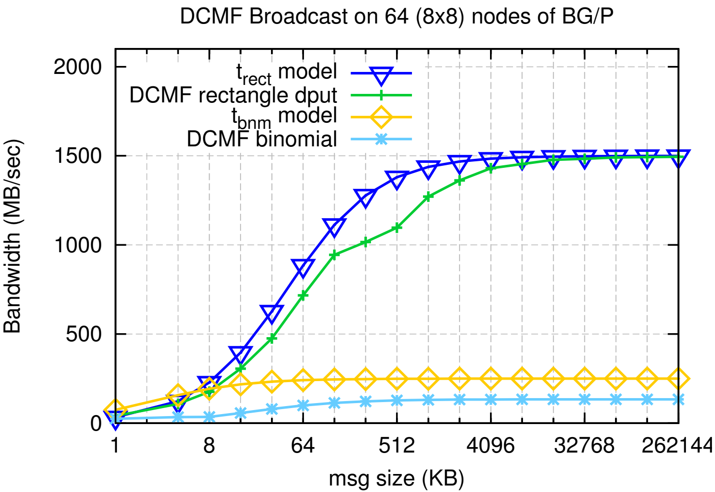
$$t_{bnm} = \log_2(P) \cdot (m/B_n + 2o + L)$$

- The root of the binomial tree sends the entire message $\log_2(P)$ times
- The setup overhead is overlapped with the path overhead
- **We assume no contention**

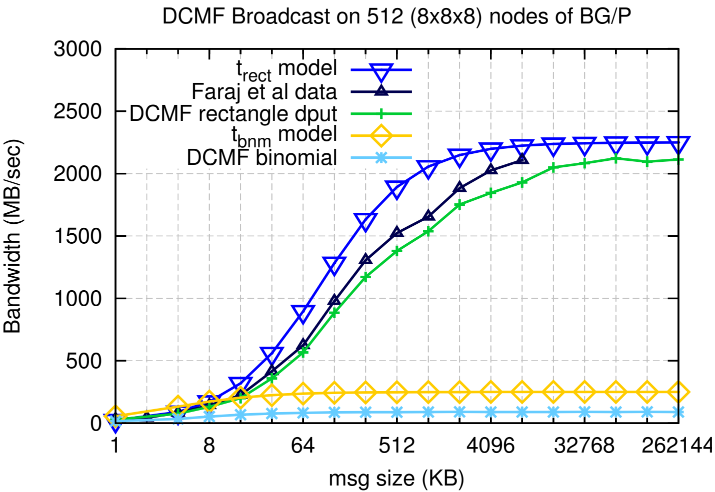
Model verification: one dimension



Model verification: two dimensions



Model verification: three dimensions

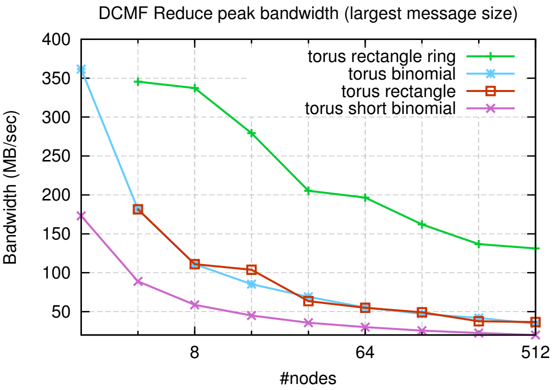


A model for rectangular reductions

$$t_{red} = \max[m/(8\gamma), 3m/\beta, m/B_n] + 2(d+1)\cdot o + 3L + d\cdot P^{1/d}\cdot(2o+L)$$

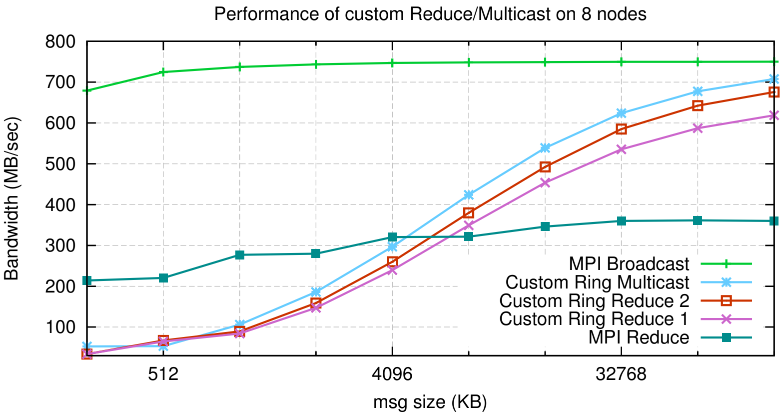
- Any multicast tree can be inverted to produce a reduction tree
- The reduction operator must be applied at each node
  - each node operates on $2m$ data
  - both the memory bandwidth and computation cost can be overlapped

Rectangular reduction performance on BG/P



BG/P rectangular reduction performs significantly worse than multicast

Performance of custom line reduction

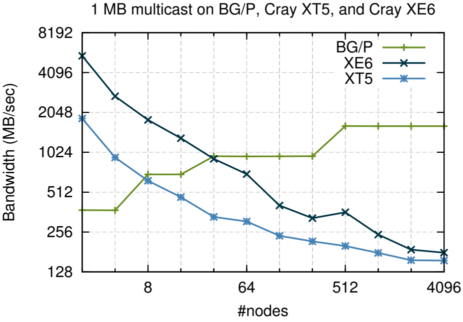


Another look at that first plot

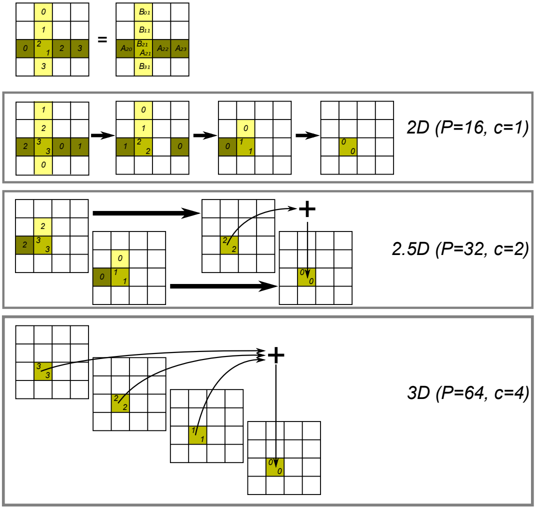
Just how much better are rectangular algorithms on $P = 4096$ nodes?

- 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?



2.5D Cannon-style matrix multiplication



Classification of parallel dense matrix algorithms

algs	c	memory (M)	words (W)	messages (S)
2D	1	$O(n^2/P)$	$O(n^2/\sqrt{P})$	$O(\sqrt{P})$
2.5D	$[1, P^{1/3}]$	$O(cn^2/P)$	$O(n^2/\sqrt{cP})$	$O(\sqrt{P/c^3})$
3D	$P^{1/3}$	$O(n^2/P^{2/3})$	$O(n^2/P^{2/3})$	$O(\log(P))$

NEW: 2.5D algorithms generalize 2D and 3D algrotihms

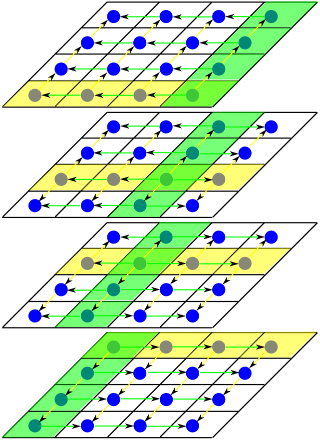
Minimize communication with

- minimal memory (2D)
- with as much memory as available (2.5D) - **flexible**
- with as much memory as the algorithm can exploit (3D)

Match the network topology of

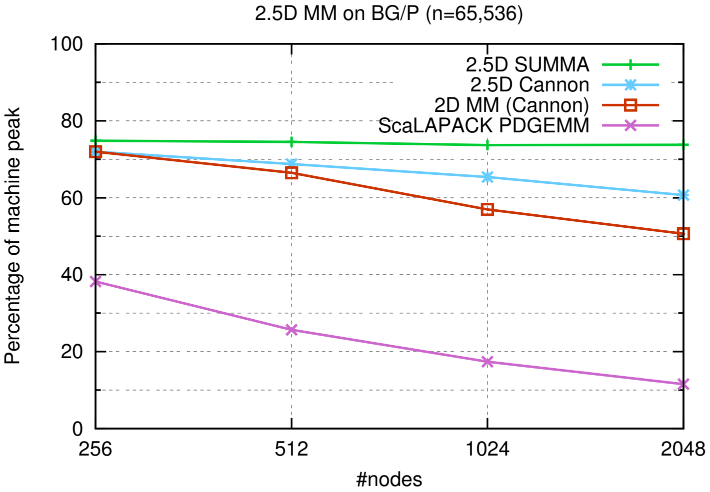
- a $\sqrt{P}$-by-$\sqrt{P}$ grid (2D)
- a $\sqrt{P/c}$-by-$\sqrt{P/c}$-by-c grid, most cuboids (2.5D) - **flexible**
- a $P^{1/3}$-by-$P^{1/3}$-by-$P^{1/3}$ cube (3D)

2.5D SUMMA-style matrix multiplication

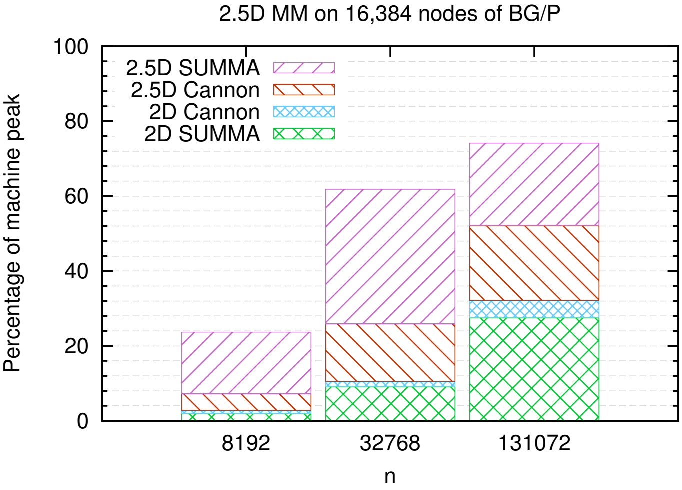


Matrix mapping to 3D partition of BG/P

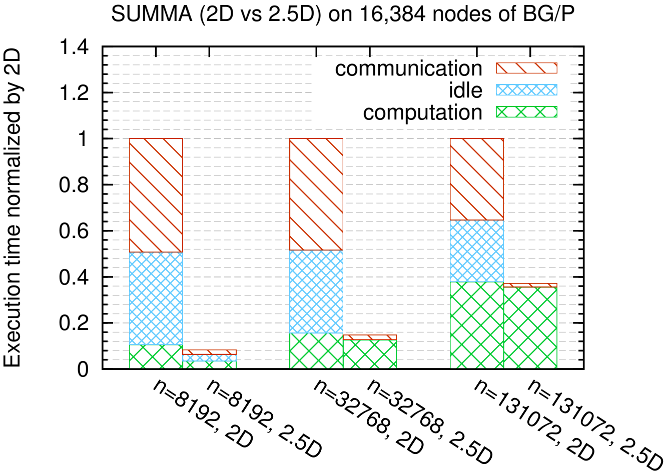
2.5D MM strong scaling



2.5D MM on 65,536 cores



Cost breakdown of MM on 65,536 cores

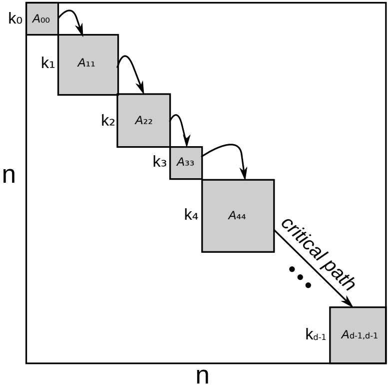


A new latency lower bound for LU

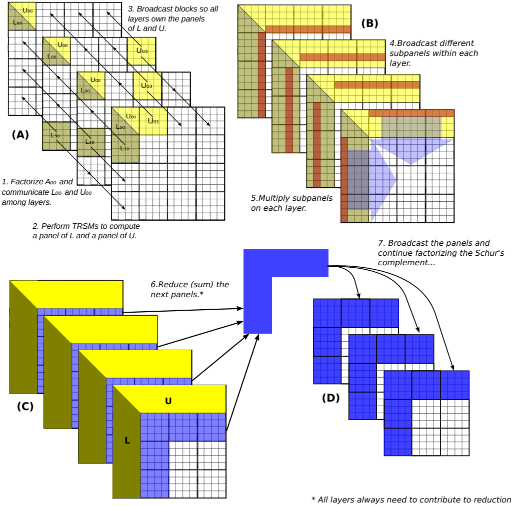
Reduce latency to $O(\sqrt{P/c^3})$ for LU?

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

No dice. But lets minimize bandwidth.

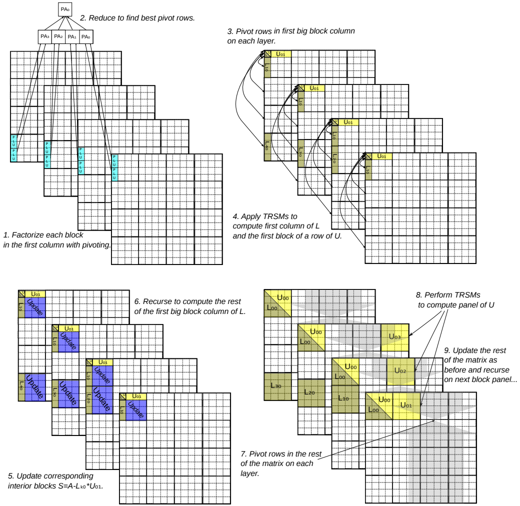


2.5D LU factorization without pivoting

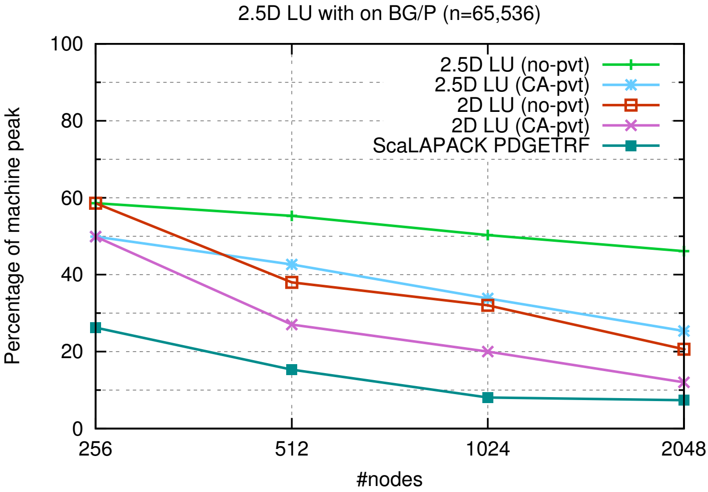


* All layers always need to contribute to reduction even if iteration done with subset of layers.

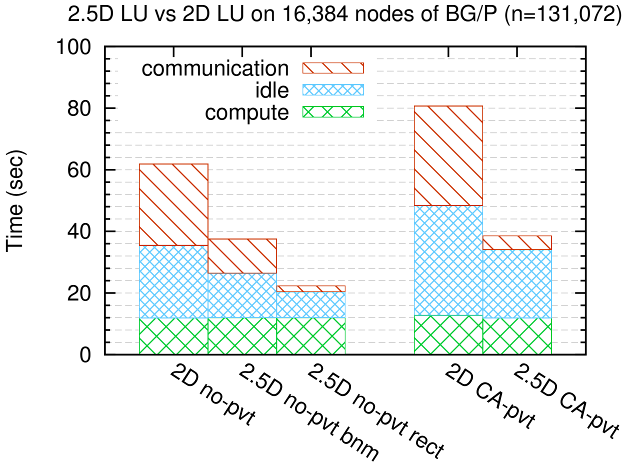
2.5D LU factorization with tournament pivoting



2.5D LU strong scaling



2.5D LU on 65,536 cores



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

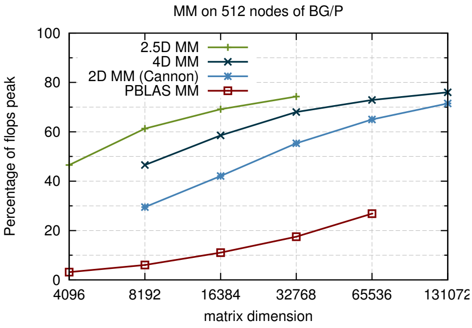
Higher-dimensional distributed MM

- 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

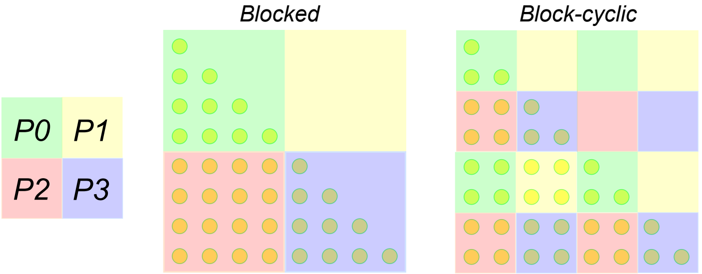
How do we map to a 3D partition *without using more memory*

- 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

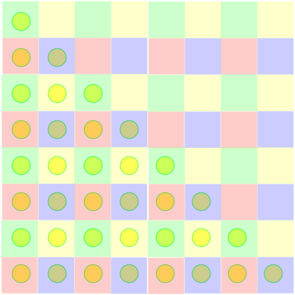
- A fully symmetric tensor of dimenson d requires only $n^d/d!$ storage
- Symmetry significantly complicates sequential implementation
  - Irregular indexing makes alignment and unrolling difficult
  - Generalizing over all partial-symmetries is expensive
- Blocked or block-cyclic virtual processor decmpositions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

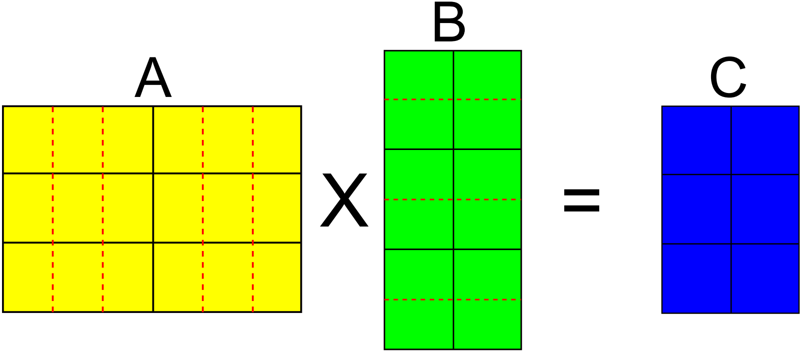
- In order to retain partial symmetry, all symmetric dimensions of a tensor must be mapped with the same cyclic phase
- The contracted dimensions of A and B must be mapped with the same phase
- And yet the virtual mapping, needs to be mapped to a physical topology, which can be any shape

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

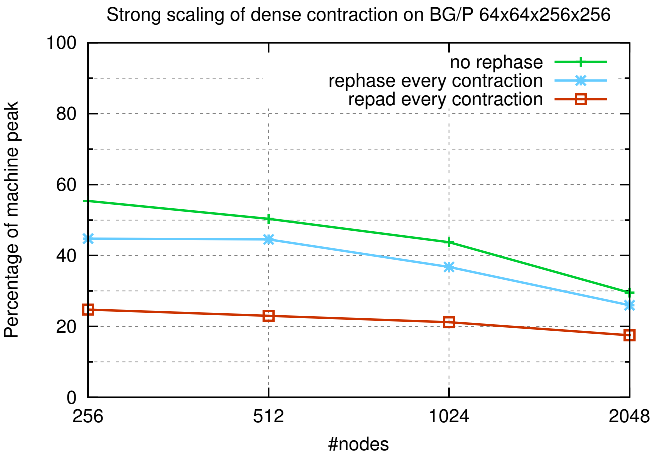
- Dense and symmetric remapping/repadding/contractions implemented
- Currently functional only for dense tensors, but with full symmetric logic
- Can perform automatic mapping with physical and virtual dimensions, but cannot unfold processor dimensions yet
- Complete library interface implemented, including 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

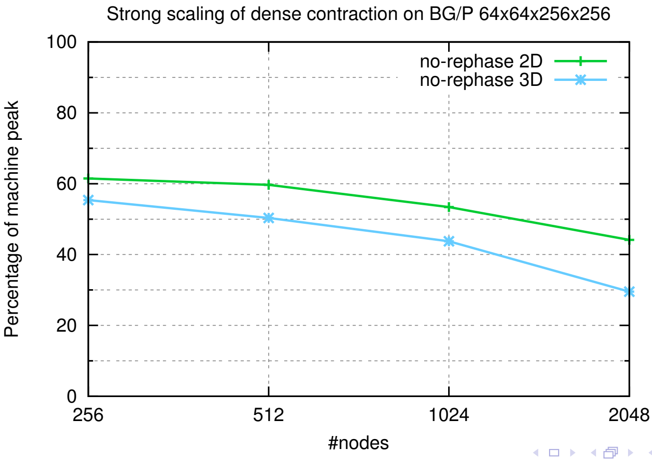
Very preliminary contraction library results

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



Potential benefit of unfolding

Unfolding smallest two BG/P torus dimensions improves performance.



Conntributions

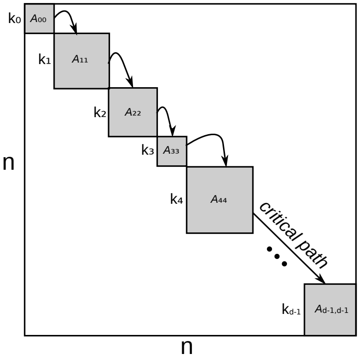
- Models for rectangular collectives
- 2.5D algorithms theory and implementation
- Using a cyclic mapping to parallelize symmetric tensor contractions
- Extending and tuning processor grid with virtual dimensions
- Automatic mapping of high-dimensional tensors to topology-aware physical partitions
- A parallel tensor contraction algorithm/library without a global address space

Conclusions and references

- Parallel tensor contraction algorithm and library seem to be the first communication-efficient practical approach
- Preliminary results and theory indicate high potential of this tensor contraction library
- papers
  - (2.5D) to appear in Euro-Par 2011, **Distinguished paper**
  - (2.5D + rectangular collective models) to appear in Supercomputing 2011

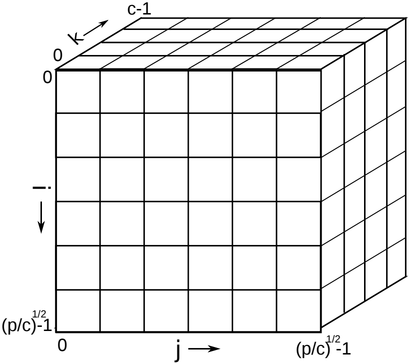
Backup slides

A new LU latency lower bound



flops lower bound requires $d = \Omega(\sqrt{p})$ blocks/messages
bandwidth lower bound required $d = \Omega(\sqrt{cp})$ blocks/messages

Virtual topology of 2.5D algorithms



2D algorithm mapping: $\left(\sqrt{P}\right) \times \left(\sqrt{P}\right)$ grid

2.5D algorithm mapping: $\left(\sqrt{P/c}\right) \times \left(\sqrt{P/c}\right) \times c$ grid **for any c**