PCSE Lecture 13

HPC Linear Algebra

Cyrus Proctor
Victor Eijkhout

April 23, 2015

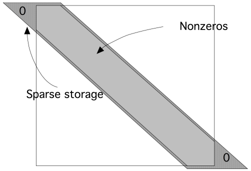
Review

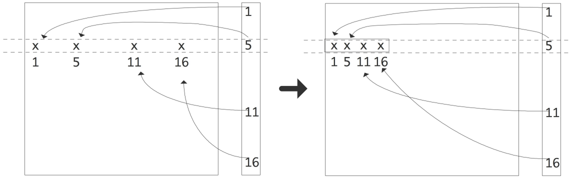
- Start out with a partial differential equation (BVP, IBVP, etc.)
- Discretize in space and time
  - Explicit
  - Implicit
- Von Neumann stability analysis
- Explicit: compute matrix vector products (MVPs)
- Implicit: compute linear system solution
- Linear system solutions
  - Direct methods
  - Iterative methods
- Iterative methods
  - Use preconditioner K to converge faster
  - MVPs, vector addition, vector inner products common
- Sparse matrix storage

Sparse Matrix Vector Products

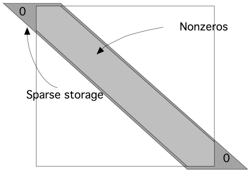
Recall the two storage types mentioned so far:

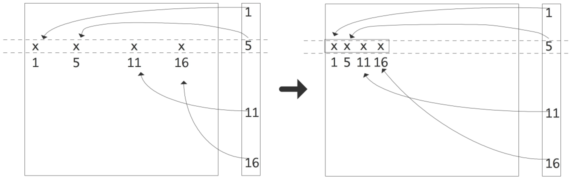
- Diagonal storage
- Compressed row storage





Sparse Matrix Vector Products





Data reuse in the SMVP:

- Similarities between dense MVP by row and sparse MVP:
  - All matrix elements are used sequentially
  - Any cache line loaded is utilized fully
- Difference for CRS MVP:
  - Indirect addressing requires loading the elements of an integer vector
  - More memory traffic for the same number of operations
  - Elements of the source vector are not loaded sequentially
  - Cacheline with source element likely to be not fully utilized

Codes dominated by sparse MVPs may only run at $\sim 5\%$ peak

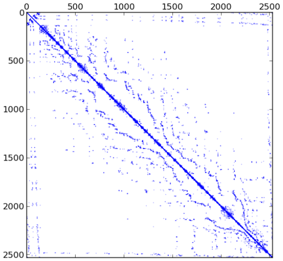
Vectorization of the Sparse MVP

In some circumstances, bandwidth and reuse are not the dominant concerns:

- Because of all the zero elements, CRS vector lengths are typically low
- GPUs/MICs expect identical operations for best efficiency
- Sparse rows in CRS are likely unequal length
- Efforts to use variations of diagonal storage have seen a revival lately

How to Improve Performance?

- If there is any regular pattern to non-zeros
- Small dense blocks
- Reduces indexing information
  - Just using 2x2 blocks gives 4X reduction in index data needed



Ordering Strategies and Parallelism

For the following methods, keep in mind that we are:

- Renumbering grid points
- Redistributing which process(es) have access to data

These strategies work well if:

- Recursion length is “large enough”
  - Dense systems usually okay
  - More work for sparse systems

All strategies are variants of Gaussian elimination.

Variable Reordering

Consider our 1D BVP tridiagonal matrix with Lexicographically ordered points:

$$\begin{pmatrix} a_{11} & a_{12} & & & & \emptyset \\ a_{21} & a_{22} & a_{23} & & & \\ & a_{32} & a_{33} & a_{34} & & \\ \emptyset & & \ddots & \ddots & \ddots & \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \end{pmatrix}$$

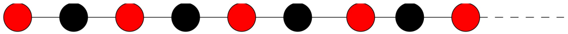
- Matrix A is known
- Vector b is known
- Vector x is unknown

How to go about maximizing available parallelism?

Red-Black Ordering

- x_j only depends on x_{j-1} and x_{j+1}

$$\begin{pmatrix} a_{11} & & & & a_{12} & & \\ & a_{33} & & & a_{32} & a_{34} & \\ & & a_{55} & & & \ddots & \ddots \\ & & & \ddots & & & \\ a_{21} & a_{23} & & & a_{22} & & \\ & a_{43} & a_{45} & & & a_{44} & \\ & & \ddots & \ddots & & & \ddots \end{pmatrix} \begin{pmatrix} x_1 \\ x_3 \\ x_5 \\ \vdots \\ x_2 \\ x_4 \\ \vdots \end{pmatrix} = \begin{pmatrix} b_1 \\ b_3 \\ b_5 \\ \vdots \\ b_2 \\ b_4 \\ \vdots \end{pmatrix}$$



Red-Black Ordering 2

Using this grouping and denoting with subscripts (b) and (r) the black and red nodes, respectively, we can write for the exact system:

$$\begin{pmatrix} D_r & U \\ L & D_b \end{pmatrix} \begin{pmatrix} x_r \\ x_b \end{pmatrix} = \begin{pmatrix} b_r \\ b_b \end{pmatrix}$$

We can turn this into an iterative update:

$$\begin{pmatrix} D_r & U \\ L & D_b \end{pmatrix} \begin{pmatrix} x_r^{k+1} \\ x_b^{k+1} \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} x_r^k \\ x_b^k \end{pmatrix} + \begin{pmatrix} b_r \\ b_b \end{pmatrix}$$

Choice of $K = A$, solution in “1 iteration”, i.e. direct solution

Red-Black Ordering 3

Instead, move L and U to right-hand side (old-iterate) for $K = D_A$ Jacobi preconditioner:

$$\begin{pmatrix} D_r & 0 \\ 0 & D_b \end{pmatrix} \begin{pmatrix} x_r^{k+1} \\ x_b^{k+1} \end{pmatrix} = \begin{pmatrix} 0 & -U \\ -L & 0 \end{pmatrix} \begin{pmatrix} x_r^{k} \\ x_b^{k} \end{pmatrix} + \begin{pmatrix} b_r \\ b_b \end{pmatrix}$$

Use (sparse) MVPs to solve:

$$D_r x_r^{k+1} = -U x_b^k + b_r$$

$$D_b x_b^{k+1} = -L x_r^k + b_b$$

Which becomes:

$$x_r^{k+1} = D_r^{-1}[-U x_b^k + b_r]$$
$$x_b^{k+1} = D_b^{-1}[-L x_r^k + b_b]$$

So, we've only changed the order that we solve the x's

Red-Black Ordering 4

Now, only move U to the right-hand side for $K = D_A + L_A$ Gauss-Seidel preconditioner:

$$\begin{pmatrix} D_r & 0 \\ L & D_b \end{pmatrix} \begin{pmatrix} x_r^{k+1} \\ x_b^{k+1} \end{pmatrix} = \begin{pmatrix} 0 & -U \\ 0 & 0 \end{pmatrix} \begin{pmatrix} x_r^{k} \\ x_b^{k} \end{pmatrix} + \begin{pmatrix} b_r \\ b_b \end{pmatrix}$$

Use (sparse) MVPs to solve:

$$D_r x_r^{k+1} = -U x_b^k + b_r$$

$$L x_r^{k+1} + D_b x_b^{k+1} = b_b$$

Which becomes (in parallel):

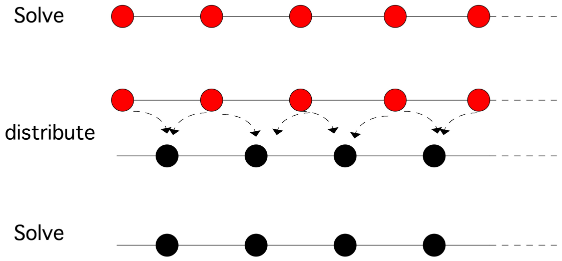
Step 1: $x_r^{k+1} = D_r^{-1}[-U x_b^k + b_r]$

Step 2: Distribute relevant parts of x_r^{k+1}

Step 3: $x_b^{k+1} = D_b^{-1}[-L x_r^{k+1} + b_b]$

Red-Black Ordering 5

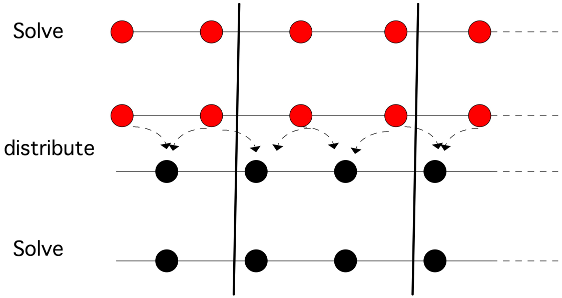
In serial:



"Distribution" step is unnecessary. All points of x_r^{k+1} are available locally.

Red-Black Ordering 6

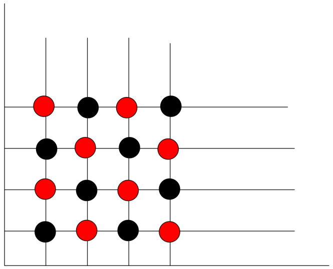
In parallel:



Distribute points of x_r^{k+1} that are needed by neighbor processes.

Red-Black Ordering 7

- Red-black ordering can be applied in 2 dimensions just as well
- Apply to points (i, j) where $1 \leq i, j \leq n$
- Successive odd points $(1, 1), (1, 3), (1, 5)$... on the first line
- Successive even points $(2, 2), (2, 4), (2, 6)$... on the second line
- Successive odd points ... on the third line

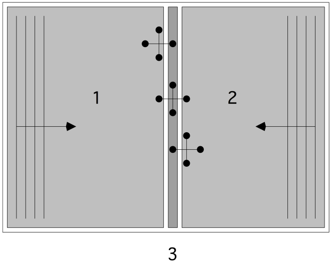


Setup is otherwise the same!

Nested Dissection Ordering A.K.A Domain Decomposition

- Initially designed as a way to reduce fill-in
- Advantageous in a parallel computing context
- Recursive process

First, split computational domain into two parts (1, 2) with a dividing strip (3):



Nested Dissection 2

- The divider must be wide enough such that the other two subdomains are not connected, i.e. decoupled.

Our resulting matrix, A^{DD} has the following structure:

$$
\left(\begin{array}{ccccc|ccccc|c}
\star & \star & & & & & & & & & 0 \\
\star & \star & \star & & & & & & & & \vdots \\
 & \ddots & \ddots & \ddots & & & & \emptyset & & & \vdots \\
 & & \star & \star & \star & & & & & & 0 \\
 & & & \star & \star & & & & & & \star \\
\hline
 & & & & & \star & \star & & & & 0 \\
 & & & & & \star & \star & \star & & & \vdots \\
 & & \emptyset & & & & \ddots & \ddots & \ddots & & \vdots \\
 & & & & & & & \star & \star & \star & 0 \\
 & & & & & & & & \star & \star & \star \\
\hline
0 & \cdots & \cdots & 0 & \star & 0 & \cdots & \cdots & 0 & \star & \star
\end{array}\right)
\begin{array}{l}
\left.\begin{array}{c} \\ \\ \\ \\ \\ \end{array}\right\} (n^2-n)/2 \\
\left.\begin{array}{c} \\ \\ \\ \\ \\ \end{array}\right\} (n^2-n)/2 \\
\left.\begin{array}{c} \\ \end{array}\right\} n
\end{array}
$$

Nested Dissection 3

In block matrix form:

$$A^{\mathrm{DD}} = \begin{pmatrix} A_{11} & \emptyset & A_{13} \\ \emptyset & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix}$$

- This gives a 3×3 block matrix structure
- Corresponds to the 3 divisions of the computational domain
- A_{12} and A_{21} are zero b/c subdomains 1 & 2 are decoupled
- Subdomain 3 contains coupling terms
- Subdomain 3 is the parent of subdomains 1 & 2
- Subdomains 1 & 2 are siblings

Nested Dissection 4

Factorize the 3×3 block matrix using an LU decomposition:

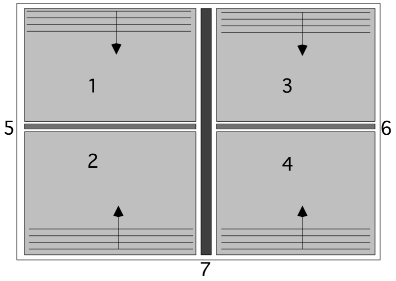
$$A^{\mathrm{DD}} = LU = \begin{pmatrix} I & & \\ \emptyset & I & \\ A_{31}A_{11}^{-1} & A_{32}A_{22}^{-1} & I \end{pmatrix} \begin{pmatrix} A_{11} & \emptyset & A_{13} \\ & A_{22} & A_{23} \\ & & S_{33} \end{pmatrix}$$

- Where $S_{33} = A_{33} - A_{31}A_{11}^{-1}A_{13} - A_{32}A_{22}^{-1}A_{23}$
- Can compute $A_{31}A_{11}^{-1}A_{13}$ and $A_{32}A_{22}^{-1}A_{23}$ simultaneously!
- The third subdomain remains sequential

Nested Dissection 5

Let's start over:

- With one division, we got a factorization that was two-way parallel
- Instead of one vertical division, try one vertical division and one horizontal division



Nested Dissection 6

This gives a block matrix form:

$$A^{\mathrm{DD}} = \left(\begin{array}{cccc|cc|c} A_{11} & & & & A_{15} & & A_{17} \\ & A_{22} & & & A_{25} & & A_{27} \\ & & A_{33} & & & A_{36} & A_{37} \\ & & & A_{44} & & A_{46} & A_{47} \\ \hline A_{51} & A_{52} & & & A_{55} & & A_{57} \\ & & A_{63} & A_{64} & & A_{66} & A_{67} \\ \hline A_{71} & A_{72} & A_{73} & A_{74} & A_{75} & A_{76} & A_{77} \end{array}\right)$$

- Subdomains 1, 2, 3, and 4 are siblings
- Subdomains 5 and 6 are parents
- Subdomain 7 is a grandparent

Nested Dissection 7

The U from the LU factorization of A^{DD} is:

$$U = \left(\begin{array}{cccc|cc|c}
A_{11} & & & & A_{15} & & A_{17} \\
& A_{22} & & & A_{25} & & A_{27} \\
& & A_{33} & & & A_{36} & A_{37} \\
& & & A_{44} & & A_{46} & A_{47} \\
\hline
& & & & S_5 & & A_{57} \\
& & & & & S_6 & A_{67} \\
\hline
& & & & & & S_7
\end{array}\right)$$

- Siblings are computed first, then parents, then grandparents
  1. Compute simultaneously sibling factorizations (more DD's?!)
  2. When siblings are finished, parents start (still parallel components)
  3. When parents finish, grandparent starts (sequential)

Nested Dissection 8

- Above process may complete until subdomains are very small
- Theoretically can go to 1×1
- Usually go to $\sim 32 \times 32$ blocks and use efficient dense solver
- Tend to see nice space savings compared to regular factorization with general 2D case
- 3D tends to have higher complexity

Nested Dissection 9

- Mapping these tasks to processors is not trivial; use task queue
- Will end up with more processors than tasks towards the end
- Last tasks are most substantial! Get free processors to help

Task Queue

```
For(all bottom level subdomains $d$)
        add $d$ to the Queue
While(Queue is not empty)
        If(a processor is idle)(assign a queued task to it)
                If(a task is finished AND its sibling is finished)
                        add its parent to the queue
```

- For distributed memory, communication is done intellegently
- Still relatively cheap compared to factorization

Grid Updates

- Sparse MVPs are largely bandwidth-bound
- Little data-reuse; indexing make locality worse
- Try to rearrange operations to lessen bandwidth demands

Consider our 1D IBVP Explicit scheme:

$$\forall_j\colon x_j^{k+1} = f(x_j^k, x_{j-1}^k, x_{j+1}^k)$$

- Assume the set $\{x_j^k\}_j$ is too large to fit in cache

Schematically:

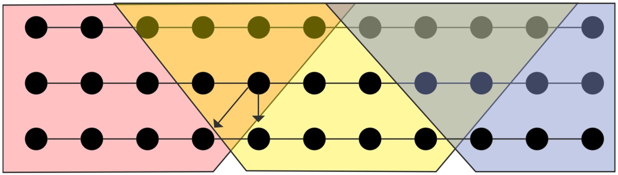
$$\begin{array}{ccc}
x_0^k & x_1^k & x_2^k \\
\downarrow\swarrow & \searrow\downarrow\swarrow & \searrow\downarrow\swarrow \\
x_0^{k+1} & x_1^{k+1} & x_2^{k+1} \\
\downarrow\swarrow & \searrow\downarrow\swarrow & \searrow\downarrow\swarrow \\
x_0^{k+2} & x_1^{k+2} & x_2^{k+2}
\end{array}$$

Grid Updates 2

$$\begin{tikzcd}
x_0^k \arrow[d] \arrow[dr] & x_1^k \arrow[dl] \arrow[d] \arrow[dr] & x_2^k \arrow[dl] \arrow[d] & {} \arrow[dl] \\
x_0^{k+1} \arrow[d] \arrow[dr] & x_1^{k+1} \arrow[dl] \arrow[d] \arrow[dr] & x_2^{k+1} \arrow[dl] \arrow[d] & {} \arrow[dl] \\
x_0^{k+2} & x_1^{k+2} & x_2^{k+2} &
\end{tikzcd}$$

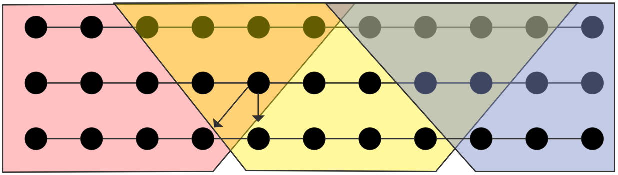
- Typically compute all x_j^{k+1} values
- Then, compute all x_j^{k+2} values using x_j^{k+1}'s
- x_j^{k+1} values are flushed from cache as they are generated
- Why not compute, not one, but two k iterations for a given j?
- x_0^{k+2} is requires x_0^{k+1} and x_1^{k+1}
  - x_0^{k+1} requires x_0^k and x_1^k
  - x_1^{k+1} requires x_0^k, x_1^k, and x_1^k

Grid Updates 3



- Suppose we only care about the final result
- Processor 0 (red) computes 4 "j" points on "k" level $(k+2)$
- For this, it needs 5 j points from level $(k+1)$
- These points need to be computed from 6 j points from level (k)
- Processor 0 needs a "ghost region" (or halo) of j size 2 instead of 1 for a single k update

Grid Updates 4



- One of the points computed by processor 1 is x_4^{k+2}
- x_4^{k+2} needs x_4^{k+1}
- x_3^{k+2} also needs x_4^{k+1}

Which processor should compute x_4^{k+1}?

- Why not both processor 0 and processor 1?

Grid Updates 5

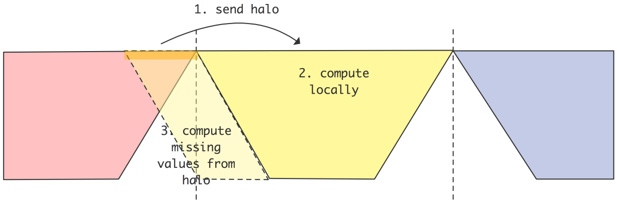
But you're doing more work?!

- Do a colored block on a single processor
- If all points in a colored block fit in cache, all $(k + 1)$ points are reused before flushing

For distributed memory computation:

- Reduces message traffic
- Neighbor processors exchange every $k = 2$ steps instead of every iteration step
- Redundant calculations in this arrangement are typically faster than an extra communication

Grid Updates 6



Communication and work minimizing strategy:

- Make algorithm more efficient
- Processor i communicates its halo points at step (k) to neighbor processors
- Neighbor processors simultaneously do the same
- Processor i, does not wait to start $(k+1)$ and then $(k+2)$ updates that already can be done locally
- Once all updates that can be done locally have finished, block for halo communication
- Compute missing values in halo

References

- Victor Eijkhout, “Introduction to High Performance Scientific Computing”
- Victor Eijkhout, “Parallel Computing for Science and Engineering”
- George Karniadakis “Parallel Scientific Computing in C++ and MPI”