(Dense) Matrix Algorithms

- *Dense* as opposed to *sparse*
  - "dense" means "arbitrary" since a dense algorithm *can* be applied to *any* matrix
  - in a "sparse" algorithm, represent a set of data items together with their locations within a structure
  - save on storage and execution time in a sparse representation
- Conventional to consider square matrices
  - simpler notation
  - easy generalisation to arbitrary (different) matrix dimensions

Topics we will consider

- Mapping of matrices onto processors
- Parallel matrix algorithms
  - matrix transpose
  - matrix $\times$ vector multiply
  - matrix $\times$ matrix multiply
  - Solution of linear equations

Striping

- *Striped partitioning by row*: matrix is divided into groups of complete rows and each group is allocated to one processor
  - *uniform striping* if all groups contain the same number of rows
  - *striped partitioning by column* similarly
- *block-striped* if the rows in each group are consecutive in the matrix
  - e.g. if a $kp \times kp$ matrix is block-striped by row on p processors, processor $i(0 \leq i \leq p-1)$ holds rows
    $ki, ki+1, \ldots, k(i+1)-1$

Cyclic Striping

- A partition is *cyclic-striped* if processor i holds all rows with index i modulo p $(0 \leq i \leq p-1)$
  - i.e. processor i holds rows $i, i+p, \ldots, i+(k-1)p$
- *block-cyclic-striped* if blocks of q rows are striped, h blocks per processor. Processor i then has rows:

$$qi, qi+1, \ldots, q(i+1)-1, q(i+p), \ldots, \\ q(i+p+1)-1 \ldots, q(i+(h-1)p), \ldots, \\ q(i+1+(h-1)p)-1$$

Checkerboarding

- Matrix is divided into rectangular blocks
  - rows *and* columns are split – no processor contains a whole row or column (except in the pathological case of striping)
  - *uniform* checkerboard uses same sized blocks
- *Block-checkerboard* if the blocks are sub-matrices, i.e. contiguous in original matrix
- *Cyclic-checkerboard* if the rows and columns in each block are selected cyclicly – cf. cyclic striping

Checkerboarding (2)

- *Block-cyclic-checkerboard* if equal sized sets of contiguous rows and columns in each block are selected block-cyclicly – cf. block-cyclic striping
- Checkerboarding naturally suited to mesh networks
  - e.g. one processor for each matrix element
  - more commonly, one processor for a sub-matrix
  - often embed logical mesh into a non-mesh physical network

Matrix Transposition

- Given matrix

$$A = [a_{ij} \mid 1 \leq i, j \leq n]$$

  require

$$A' = [a_{ji} \mid 1 \leq i, j \leq n]$$

- Need to exchange the corresponding $n^2 - n$ off-diagonal elements
  $\Rightarrow (n^2 - n)/2$ exchanges
  $\simeq n^2/2$ time-complexity on a single processor

Checkerboard Partitioning

- First consider a mesh implementation with one element per processor
- i, j element moves up to its diagonal and then across the same number of hops to j, i position
- Need to synchronise the communication because of multiple transmissions on each link

Run-time on a Mesh

- All communications are concurrent point-to-point simple message transfers
- Maximum distance is $2(n-1)$ hops
- $\Rightarrow$ latency is $2(n-1)(t_s + t_h + t_w)$ assuming one-word elements and separate SF communications

Mesh of $p < n$ Processors

- Assume p is a perfect square and that n is an integer multiple of $\sqrt{p}$
- Then the matrix is checkerboarded as a $\sqrt{p} \times \sqrt{p}$ block-matrix of $(n/\sqrt{p}) \times (n/\sqrt{p})$ blocks
- Algorithm is now:
  - transpose each block locally $\Rightarrow$ time complexity $\simeq n^2/2p$
  - transpose blocks as above $\Rightarrow$ communication latency $\simeq 2(\sqrt{p}-1)(t_s + t_w n^2/p)$

Parallel Run-Time

- Parallel run-time with store-and-forward routing is

$$\begin{aligned} T_p &= n^2/2p + 2(\sqrt{p}-1)(t_s + t_w n^2/p) \\ &\simeq n^2/2p + 2t_s\sqrt{p} + 2t_w n^2/\sqrt{p} \end{aligned}$$

  treating the per-hop terms t_h as negligible
- So the cost is $C_p = pT_p = \Theta(\sqrt{p}n^2)$
- So *not cost optimal*

Hypercube

Matrix may be transposed by transposing a block-matrix of its transposed blocks – as above

- Apply this procedure recursively to transpose the blocks – *recursive transposition algorithm*
- Base case is a 2×2 matrix
- $n \times n$ matrix naturally maps onto a p-hypercube
  - Suppose n and p are both powers of 2, $p \leq (n/2)^2$ and $\log p$ even ($p = 4, 16, 64, \ldots$)
  - Base cases are sub-matrices with n^2/p elements on a p-hypercube (≥ 4 elements)

Hypercube Recursive Algorithm

- 1. First, allocate matrix blocks to 4 sub-cubes:
  - divide the matrix into 4 blocks
  - partition the hypercube into 4 sub-cubes – i.e. split in two twice
  - allocate 1 block to each sub-cube, swapping the off-diagonal blocks
- 2. Repeat the algorithm for each block on each sub-cube until the sub-cubes each contain one processor
  - $(\log p)/2$ steps, since reduce dimension by two each step

Hypercube Recursive Algorithm (2)

- In each step, the mesh algorithm on 2×2 *block*-matrices is applied
  - At each step, corresponding pairs of processors in different sub-cubes exchange data through another corresponding processor in a diagonal sub-cube
  - After each step, sizes of block-matrices transposed in parallel are quartered
- 3. Finally, transpose the $(n/\sqrt{p}) \times (n/\sqrt{p})$ blocks locally in each processor

Hypercube Recursive Algorithm (3)

- Size of message representing swapping blocks is always n^2/p
- Hence parallel run time is:

$$T_p = n^2/2p + (t_s + t_w n^2/p) \log p$$

- Again, *not cost optimal*

Block Striping

- First suppose each of p processors has one row of the $p \times p$ matrix
  - processor i must send its jth row element to processor j $(1 \leq i \neq j \leq p)$
  - *all-to-all personalised communication*
- Now let each processor hold n/p rows of a $n \times n$ matrix
  - must send n/p contiguous elements from each row to every other processor
  - parallel communication of $n/p \times n/p$ block-matrices, i.e. messages of size $m = n^2/p^2$

Block Striping (2)

- Then every processor transposes each of its p blocks
  - $\Rightarrow$ time complexity $p.n^2/(2p^2) = n^2/(2p)$
  - *familiar*?!
- Best parallel run time is on a hypercube with cut-through (fastest ATAP communication)
  - $\Rightarrow T_p = \frac{n^2}{2p} + t_s(p-1) + \frac{t_w n^2}{p} + \frac{t_h p \log p}{2}$
  - cost-optimal ??

Matrix $\times$ Vector

Compare row-wise striping with checkerboarding

- $n \times n$ matrix $\cdot$ vector of n components
- Note that the numbers of processors are bounded by the sizes of the vector (n) and matrix (n^2) respectively
- Consider the relative performance and scalabilities of each partitioning on different interconnection topologies

Row-wise striping

First consider one row + one component of the vector per processor

- Every processor requires the whole vector, so we need
  - all-to-all broadcast of each processor's vector-component to all the others
  - then processor i performs the dot-product of the ith row with the vector
  - leaves the ith component of the result-vector in processor i

Row-wise striping (2)

- Parallel run-time is $\Theta(n)$ because ATA communication and vector dot-product are
  - *must be cost-optimal*
- We'll consider *block-striping* the matrix on fewer processors
  - can be sure of a cost-optimal algorithm
  - recall cost-optimality is always preserved by increasing the grain size

$p = n/k$ Processors

- Block-stripe with k rows + k vector components per processor
- ATA broadcast of k elements in each message. Latency:
  - $\simeq 2t_s\sqrt{p} + t_w k(p-1) \simeq 2t_s\sqrt{p} + t_w n$ on a mesh
  - $\simeq t_s \log p + t_w k(p-1) \simeq t_s \log p + t_w n$ on a hypercube
- Followed by local computation of k dot-products in each processor, i.e. $\Theta(nk)$ computation time

Parallel Run-Time

- Thus, the asymptotic parallel run-time is
  - $n^2/p + 2t_s\sqrt{p} + t_w n$ (mesh)
  - $n^2/p + t_s \log p + t_w n$ (hypercube)
- Cost-optimal for $p \leq \Theta(n)$
  - always holds
  - even for fine-grain striping of one row per processor, as we anticipated

Scalability

- Recall that the overhead is $O_p = pT_p - W$ for problem size W on p processors. So:
  - $O_p = 2t_s p\sqrt{p} + t_w np$ on a mesh
  - $O_p = t_s p \log p + t_w np$ on a hypercube
- Isoefficiency is given by the function $W(p)$ that satisfies the equation

$$W(p) = KO_p$$

  where $K = E/(1-E)$ at given efficiency E
- Note that $n \geq p \Rightarrow W = \Omega(p^2)$ gives a *lower bound*

Scalability (2)

- Since $W = \Theta(n^2)$ we solve

$$\Theta(n^2) = 2t_s p\sqrt{p} + t_w np$$

  - Thus $\Theta(n) = t_w p + o(p)$ since $p \leq \Theta(n)$
  - same argument and result for hypercube
- So $W = \Theta(p^2)$ *for mesh and hypercube*
- *Hence must increase problem size (here size of matrix) as the square of the number of processors to maintain efficiency*
  - e.g. double dimension of matrix for double the number of processors

Column-wise striping

- Each processor holds one column of matrix + one vector-component ..... which?
- Each processor performs n multiplications
- Then all-to-one (single-node) accumulation, reducing with + on the columns
- Result-vector ends up in the accumulating node
- $\Theta(n)$ complexity
- Similar complexity to row-wise striping for $p \leq n$ processors

Checkerboard Partitioning

Assume first that $p = n^2$ for an $n \times n$ matrix

1. Send vector component i to processor (i, i)
   - may be set up this way initially
   - otherwise, assume vector is sent via n concurrent simple message transfers $\Rightarrow$ latency $\Theta(n) \mid \Theta(\log n)$ on mesh | hypercube
2. Parallel one-to-all broadcast of vector components in the diagonal processors to whole columns $\Rightarrow$ latency $\Theta(n) \mid \Theta(\log n)$ on mesh | hypercube

Checkerboard Partitioning (2)

3. Local scalar multiplications (as in columnwise striping), constant computation time
4. Concurrent single-node accumulation, reducing with + across columns (as in columnwise striping) $\Rightarrow$ latency $\Theta(n) \mid \Theta(\log n)$ on mesh | hypercube

- Net performance:
  - latency $\Theta(n)$, cost $\Theta(n^3)$ on a mesh
  - latency $\Theta(\log n)$, cost $\Theta(n^2 \log n)$ on a hypercube
- *Not cost-optimal*

$p < n^2$ Processors

- Assume p is a perfect square such that $n = k\sqrt{p}$
- Exactly the same algorithm works except that:
  - message size is k in each communication step, corresponding to k components per processor
  - computation step involves the dot product of a $k \times k$ matrix and k-vector $\Rightarrow k^2$ products and $k(k-1)$ sums

Performance

- Parallel run-time on (e.g.) a *mesh with cut-through* is then:

$$\begin{aligned} T_p &= n^2/p + t_s + t_w n/\sqrt{p} + t_h\sqrt{p} \\ &\quad +(t_s + t_w n/\sqrt{p})\log\sqrt{p} + t_h\sqrt{p} \\ &\quad +(t_s + t_w n/\sqrt{p})\log\sqrt{p} + t_h\sqrt{p} \\ &\simeq n^2/p + t_s\log p + (t_w n\log p)/\sqrt{p} + 3t_h\sqrt{p} \end{aligned}$$

- Cost-optimal if $p\sqrt{p} \leq \Theta(n^2)$ i.e. $n \geq \Theta(p^{3/4})$ (*not obvious*!)
- Isoefficiency function is, asymptotically, $\Theta(\max(p(\log p)^2, p^{3/2})) = \Theta(p^{3/2})$

Striping vs. Checkerboarding

- Computation times for an $n \times n$ matrix and p processors are the same, viz n^2/p
  - This is simply because we did not consider non-communication overheads
- Communication times for the mesh networks we considered are:
  - $2t_s\sqrt{p} + t_w n$ (row-wise striping)
  - $t_s\log p + (t_w n\log p)/\sqrt{p} + 3t_h\sqrt{p}$ (checkerboard)
- Thus, block-checkerboarding is faster

Striping vs. Checkerboarding (2)

- Similarly, checkerboarding has a smaller asymptotic isoefficiency function:
  - $\Theta(p^2) > \Theta(p^{3/2})$
  - hence checkerboarding more scalable
- Similar results for a hypercube (exercise)
- But does this *really* mean the checkerboard algorithm is better?

Striping vs. Checkerboarding (3)

- we've changed two things in our comparison
  - *partitioning strategy* – row striping vs. checkerboarding, and
  - *communication switching mechanism* – store-and-forward vs. cut-through
- But remember, for all-to-all communication on *all* the networks we've considered, store-and-forward achieves the optimal *data transmission time term*, viz. $(p-1)mt_w$
- This term may or may not dominate ...

Matrix × Matrix

- Simple serial algorithm requires calculation of n^2 vector dot-products
  - asymptotic serial run-time is n^3
  - in fact, best known serial run-time is achieved by Strassen's algorithm ("divide-and-conquer" type) which has asymptotic complexity $n^{2.8}$
- But we'll take "best" serial run-time to be $T_1 = \Theta(n^3)$ for simplicity, and because ...

In support of T_1 ...

If we were to compare parallel run-time T_p with Strassen's algorithm as the "best serial run-time':

- We would have to use Strassen's algorithm itself for local sub-matrix multiplications (at least) for a fair comparison;
- Also note that Strassen's algorithm tends to be very unstable numerically, so often not usable in practice

Simplest Parallel Algorithm

- To multiply two $n \times n$ matrices A and B with result $C = A \cdot B$, use *block-checkerboard partitioning*
- Processor $(i, j),\ 0 \leq i, j \leq \sqrt{p} - 1$, holds the $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ block-matrices A_{ij}, B_{ij} defined by:

  $$\{a_{hk} \mid i\sqrt{p} \leq h < (i+1)\sqrt{p}, j\sqrt{p} \leq k < (j+1)\sqrt{p}\}$$
  $$\{b_{hk} \mid i\sqrt{p} \leq h < (i+1)\sqrt{p}, j\sqrt{p} \leq k < (j+1)\sqrt{p}\}$$

  and computes the result $C_{ij} = A_{ij} \cdot B_{ij}$
- To compute C_{ij}, need all sub-matrices A_{ik} in *row* i and all B_{kj} in column j $(0 \leq k \leq \sqrt{p} - 1)$

The Algorithm

1. All-to-all broadcast of the sub-matrices A_{ik} *in each row* i
2. All-to-all broadcast of the sub-matrices B_{kj} *in each column* j
3. Computation *in each processor* (i, j) of vector dot-products of:
   - the row vectors obtained by catenating the rows of $A_{i0}, \ldots, A_{i(\sqrt{p}-1)}$

   and

   - the column vectors obtained by catenating the columns of $B_{0j}, \ldots, B_{(\sqrt{p}-1),j}$

Computation time

- Computation of dot-products is the same for any network topology
  - $(n/\sqrt{p})^2 = n^2/p$ dot-products of vectors of n components
  - $\Rightarrow n^3/p$ computation time

Mesh

- Communication comprises two all-to-all broadcasts amongst $\sqrt{p}$ processors (in the rows and columns)
- Message size is n^2/p elements (submatrices)
- Hence communication latency (neglecting t_h) is $2\{t_s\sqrt{p} + t_w(n^2/p)(\sqrt{p}-1)\} \simeq 2(t_s\sqrt{p} + t_w n^2/\sqrt{p})$ for large p
- So parallel run-time is $T_p = n^3/p + 2t_s\sqrt{p} + 2t_w n^2/\sqrt{p}$
- Cost, $C_p = n^3 + 2t_s p^{3/2} + 2t_w n^2\sqrt{p}$ $\Rightarrow$ cost-optimal if $p = O(n^2)$

Isoefficiency Function (Mesh)

- $O_p = 2t_s p^{3/2} + 2t_w W^{2/3}\sqrt{p}$ and so the isoefficiency function may be
  - $W = 2Kt_s p^{3/2}$ (first term only)
  - or $W = 2Kt_w W^{2/3}\sqrt{p}$
  - $\Rightarrow W = 8K^3 p^{3/2} t_w^3$ (second)
- Thus isoefficiency is $\Theta(p^{3/2})$

Hypercube

- Same reasoning, but communication latency is $2\{t_s \log\sqrt{p} + t_w(n^2/p)(\sqrt{p}-1)\}$
- Hence parallel run-time is (for large p) $T_p = n^3/p + t_s \log p + 2t_w n^2/\sqrt{p}$
- So cost, $C_p = n^3 + t_s p \log p + 2t_w n^2\sqrt{p}$
- $\Rightarrow$ cost-optimal if $p = O(n^2)$
- Same as mesh asymptotically

Isoefficiency Function (Hypercube)

- $O_p = t_s p \log p + 2t_w W^{2/3}\sqrt{p}$ and so the isoefficiency function may be
  - $W = K t_s p \log p$ (first term only)
  - or $W = 2Kt_w W^{2/3}\sqrt{p}$
  - $\Rightarrow W = 8K^3 p^{3/2} t_w^3$ (second)
- But $p \le n^2$ and so $p^{3/2} \le n^3$. Thus again isoefficiency is $\Theta(p^{3/2})$ since this is $\Omega(p \log p)$

Space Optimisations

- Disadvantage of this algorithm is a *high memory requirement*
  - $\sqrt{p}$ blocks at each processor
  - $\Rightarrow$ total memory needed $= \Theta(n^2\sqrt{p})$ as opposed to $\Theta(n^2)$ for the serial algorithm
- Space optimisations can be obtained by *rotating* block submatrices so that parts of the computation at each processor can be done after each alignment of blocks

Cannon's and Fox's Algorithms

- In Cannon's algorithm, blocks are rotated horizontally and vertically by "appropriate amounts"
  - left (circular) shift block A_{ij} through i positions
  - up shift block B_{ij} through j positions
  - accumulate block C_{ij} by adding block matrix products
- Fox's algorithm combines $\sqrt{p}$ successive one-to-all row broadcasts with $\sqrt{p}$ single step upwards shifts

A Time Optimisation (DNS)

- Dekel-Nassimi-Sahni algorithm improves parallel run-time by using up to n^3 processors
- Assign each of the n^3 scalar multiplications to a separate processor, say P_{ijk} if processors are organised in n $n \times n$ planes
- Then sum the values (products of previous step) in each column of the plane
  - P_{ijk} holds A_{ik} and B_{kj} and multiplies them
  - column sum is therefore $A_{i\cdot} \cdot B_{\cdot j} = C_{ij}$

Abstract Implementation

- On a CREW PRAM (abstract machine), assume A and B are distributed over plane 0 – i.e. P_{ij0} has A_{ij} and B_{ij}
- All processors in the other planes fetch their data in constant time
- Multiplications take unit time (in parallel) and the additions can be done in $\log n$ steps
- Hence $\log n$ (asymptotic) complexity
- Not cost-optimal with CREW
- Cost-optimal with CRCW if additions are done by writes reducing with +

Hypercube Implementation

- Need to move data physically, in contrast to the PRAM
- Assume $n = 2^d$ and that the planes consist of n sub-cubes – connected at corresponding nodes, as in the recursive definition of a hypercube
- DNS algorithm has n^3 scalar multiplications in parallel (constant time) + three communication steps:

Hypercube Implementation (2)

1. one-to-one communication of each column (*c*) of A and each row (*r*) of B to respective planes (*c*,*r*)
   - A_{ij} goes to P_{ijj}
   - B_{ij} goes to P_{iji}
2. one-to-all broadcast along rows (for A) and columns (for B) in each plane above 0
   - A_{ij} broadcast to P_{ikj}
   - B_{ij} broadcast to P_{kji}
3. single node accumulation in the third dimension, reducing with +

Fewer Than n^3 Processors

- Each step has latency $\Theta(\log n) \Rightarrow T_p = \Theta(\log n)$ in the above algorithm, *so not cost-optimal*
- Consider, therefore, $p = q^3$ processors, where $q < n$ and q divides n
- Partition the matrix into p blocks of size $(n/q) \times (n/q)$
- DNS algorithm is as above except that operations are now on submatrices – i.e. matrix multiplication and addition

Hypercube Implementation

- On a hypercube, we find, ignoring the relatively small contribution from the one-to-one communication in the first step (to plane 0)

$$\begin{aligned} T_p &\simeq (n/q)^3 + 3t_s \log q + 3t_w (n/q)^2 \log q \\ &= n^3/p + t_s \log p + t_w (n^2/p^{2/3}) \log p \end{aligned}$$

- Cost-optimal if $n^3 = \Omega(p(\log p)^3)$
- Isoefficiency function is $\Theta(p(\log p)^3)$