18.435/6.848 Theory of Parallel and VLSI Computation	**Fall 1997**

Solutions to Assignment #3

TA: Daniel Lewin *Prof. Leighton*

Problem 1

The inputs are an $N \times N$ matrix A, and a vector b, and we wish to obtain the result $c = Ab$. We break up the matrix into $\sqrt{\log N} \times \sqrt{\log N}$ blocks. This way we have the same number of blocks as leaves in the reduced MOT. Thus, the leaf in the MOT at (i, j) holds inputs $a_{k,l}$ for $(i-1)\sqrt{\log N} + 1 \leq k \leq i\sqrt{\log N}$ and $(j-1)\sqrt{\log N} + 1 \leq l \leq j\sqrt{\log N}$. Now, b is divided into $N/\sqrt{\log N}$ chunks of size $\sqrt{\log N}$ each. The i^{th} chunk (comprising of $b_{(i-1)\sqrt{\log N}+1}, b_{(i-1)\sqrt{\log N}+2} \cdots b_{i\sqrt{\log N}}$) is input at the root of the i^{th} column tree in pipelined fashion, whence they are broadcast to the leaves. On receiving its j^{th} input, the leaves multiplies this value by the jth column block of its input matrix. Hence in $O(\log N)$ steps, the $(i.j)$ leaf of the MOT holds $a_{k,l}b_l$ for $(i-1)\sqrt{\log N}+1 \leq k \leq i\sqrt{\log N}$ and $(j-1)\sqrt{\log N}+1 \leq l \leq j\sqrt{\log N}$. Next, each leaf sums up its blocks column-wise to get $\sqrt{\log N}$ partial sums. This takes an additional $O(\log N)$ time. Next these partial sums are passed up its row tree in a pipelined fashion. Each internal row tree node on receiving two inputs simply sums them up and passes it up to its parent. Thus the final result is output from the roots of the row tree node, with the i^{th} root outputting $c_{(i-1)\sqrt{\log N}+1}, c_{(i-1)\sqrt{\log N}+2} \cdots c_{i\sqrt{\log N}}$.

For matrix multiplication, we use a 3-dimensional MOT of size $\frac{N}{\sqrt{\log N}} \times N \times \frac{N}{\sqrt{\log N}}$. Input the values $A_{i'j}$ for $(i-1)\sqrt{\log N} \leq i' \leq i\sqrt{\log N}$ into the root of the tree whose leaves are defined by the line i, j $(1 \leq i \leq \frac{N}{\sqrt{\log N}}, 1 \leq j \leq N)$. Similarly, input $B_{jk'}$ for $(k-1)\sqrt{\log N} \leq k' \leq k\sqrt{\log N}$ into the root of the tree whose leaves are defined by the line j, k $(1 \leq j \leq N, 1 \leq k \leq \frac{N}{\sqrt{\log N}})$.

Next, the root sends its values down to the leaves. Thus, the leaf (i, j, k) receives $2\sqrt{\log N}$ values $A_{i'j}$ and $B_{jk'}$ for $(i-1)\sqrt{\log N} \leq i' \leq i\sqrt{\log N}$ and $(k-1)\sqrt{\log N} \leq k' \leq k\sqrt{\log N}$. Then, leaf (i, j, k) computes all the $\log N$ products, and these are sent up the dimension-2 trees in a pipelined fashion. The root of the dimension-2 tree i, k receives the $\log N$ sums which can be easily verified to be the entries of the result matrix $A \cdot B$.

Another way to multiply two $N \times N$ matrices will use N "layers" of the network used for the matrix-vector multiplication. Since we can think of multiplying matrix A with matrix B equivalently as N matrix-vector multiplications (multiplying matrix A with b_i the i^{th} column of B). The i^{th} layer of our network will perform this operation in $O(\log N)$ steps.

Problem 2

The inputs are an $N \times N$ matrix A, and a result vector b, where we wish to solve for x in the equation $Ax = b$. We break up the matrix into $\sqrt{\log N} \times \sqrt{\log N}$ blocks. This way we have the same number of blocks as leaves in the reduced MOT. Thus, the leaf in the MOT at (i, j) holds inputs $a_{k,l}$ for $(i-1)\sqrt{\log N}+1 \leq k \leq i\sqrt{\log N}$ and $(j-1)\sqrt{\log N}+1 \leq l \leq j\sqrt{\log N}$. The leaves $(i, N/\sqrt{\log N})$ (the last column) hold $\sqrt{\log N}$ b values as well.

The first step is to determine the next pivot. This is accomplished by finding the maximum absolute value in A which has not already been part of a pivot row. Each time a pivot row is selected as a pivot in the algorithm, its values are marked so that the row can no longer be selected. To find the maximum, each leaf finds the maximum value of the absolute values of its unmarked A values (since there are $\log N$ such values this takes $O(\log N)$ time). Then using the row trees, the mesh calculates the maximum of these values in each row ($O(\log N)$ time). The row roots then pass their maximum value to the leaf in the first column, and this column tree then finds the maximum over all the rows in $O(\log N)$ time. The values being passed around are tagged with their origin, so we know which row/column the new pivot value came from.

Let k be this matrix row, and m this matrix column so $a_{k,m}$ is the pivot value. Then the MOT broadcasts $a_{k,m}$ to all values in the pivot row; this takes $O(\log N)$ time. The pivot row then normalizes its values by replacing $a_{k,i}$ with $a_{k,i}/a_{k,m}$, and the last column also replaces b_k with $b_k/a_{k,m}$. This having been done, the pivot row then uses the column trees to broadcast its values to the other rows in $O(\log N)$ time. Now, the MOT must zero values in matrix column m (other than the pivot row's value). This is accomplished by subtracting $a_{i,m}$ multiples of the pivot row from row i for each i. Thus, the MOT broadcasts $a_{i,m}$ to the rest of row i. This is done using the row trees. Since there are $\sqrt{\log N}$ rows in the matrix per row tree, $\sqrt{\log N}$ values must be broadcast from the same leaf. This can be done by pipelining the values in $O(\log N + \sqrt{\log N}) = O(\log N)$ time. Next, locally, each leaf updates its values $a_{i,j}$ as follows: $a_{i,j} = a_{i,j} - a_{i,m}a_{k,j}$ and $b_i = b_i - a_{i,m}b_k$ where $a_{k,j}$ is the normalized value of a in the pivot row and b_k is the normalized value of b in the pivot row. This requires $\log N$ local computations since that many values need to be updated in each leaf. The above algorithm is repeated N times at which point the elimination is complete.

The results are now the updated b_i's. In order to output them in the correct order, we simply need the leaf containing b_i to remember the column the pivot was located in when b_i's row was selected as the pivot row. The row trees are then used to do a pipelined broadcast, to send the b_i to the appropriate column in the MOT in $O(\log N)$ steps. The column trees then pass their b_i's to the root. Since there are only $\sqrt{\log N}$ values to be passed to the root of each column tree, this takes time $O(\log N)$ despite collisions which may occur. Thus, Gaussian elimination using total pivoting takes $NO(\log N) = O(N \log N)$ time.

Problem 3

We will use a recursive approach to solve the image connected components problem (ICCP) by

1 Partition the array into four quadrants.

2 Solve the ICCP for each of the four quadrants.

3 Using the connected components algorithm for graphs (GCCP) described in class, determine which components are connected along the border of the four quadrants.

4 Passing the information discovered along the borders to inform the pixels "inside" each quadrant about the new labeling. This step will also be performed recursively.

We will first describe step three of the algorithm, which will be performed in two phases. In the first phase, we will merge components along the vertical boundary. The next phase will merge components along the horizontal boundary. Since there are at most $\sqrt{N}$ different components on either side of the border, we can apply the GCCP algorithm on these components. The GCCP algorithm will take as input the following graph G: The vertices of G will be various components along the vertical border. An edge in G connects two vertices iff the nodes represented by the components are adjacent across the vertical boundary. The adjacency matrix for G is setup by sorting the components incident to the boundary. These will then correspond to rows and columns of the adjacency matrix. The processors along the boundary then send a 1 to the appropriate entry of the matrix if their component is adjacent to the other component (otherwise it sends a 0). We will omit details here, but it is important to note that since the adjacency matrix for G has size $\sqrt{N} \times \sqrt{N}$, this operation can be performed in time $O(\log N)$. We then run GCCP on G which takes $O(\log^2 N)$ time. Now the next phase begins in which components along the horizontal boundary are merged. This can be done in exactly the same manner as above. However, we may need to update labels, since if a component (say with label A) gets a new label (say B) during the vertical merge phase, and if A lies along both the horizontal and vertical borders of the quadrant, then A is changed to B along the horizontal border before the next phase begins. After GCCP is run along horizontal border, the same operation is performed along the vertical boundary. This update can be performed in $O(\log N)$ steps. Thus, step three takes time $O(\log^2 N)$.

To finish our description of the algorithm, we need to show how step four is implemented. Recall we need to update the internal pixels of the "new" labels (obtained along the vertical and horizontal borders). This is done by recursively breaking the problem into smaller and smaller quadrants and updating along the borders of these smaller quadrants. Notice that since the four smaller problems only double the length of the border at each step of the recursion, we can use the same approach as described in the previous paragraph to update the labels along the borders for each quadrant. This step can be performed in time $O(\log N)$. Thus, the total time taken to perform the update of labels for N internal points $Q(N)$ can be upper-bounded by $Q(N) = Q(N/2) + O(\log N) = O(\log^2 N)$.

Thus, the total running time for ICCP is:

$$\begin{aligned} T(N) &= T(N/4) + Q(N) + O(\log^2 N) \\ &= T(N/4) + O(\log^2 N) \\ &= O(\log^3 N) \end{aligned}$$

Problem 4

When defining distance between two pixels in an image we have two choices. We can use the Manhattan distance (or L_1 metric) which defines the distance between pixels (i_1, j_1) and (i_2, j_2) to be $|i_1 - i_2| + |j_1 - j_2|$. This gives rise to a fairly straightforward recursive solution which will not be described here. Alternatively we can use the Euclidean distance (or L_2 metric) which defines the distance to be $\sqrt{(i_1 - i_2)^2 + (j_1 - j_2)^2}$. This requires us to be a bit more clever, but we will now describe an algorithm to compute the nearest 1-pixel for each pixel in a $\sqrt{N} \times \sqrt{N}$ image using the Euclidean metric on a $\sqrt{N} \times \sqrt{N}$ MOT in $O(\log^2 N)$ steps.

The algorithm operates in two phases. During the first phase we build up larger and larger regions such that for each region each boundary pixel of that region knows its nearest 1-pixel within that region. Thus at the end of the first phase, each boundary pixel of the image will know its nearest 1-pixel over the entire image — that is each boundary pixel will be done. In the second phase we work our way back down filling in nearest 1-pixel values over the entire image for the boundaries of successively smaller and smaller regions.

We can describe the first phase as a recursive process. We partition the $\sqrt{N} \times \sqrt{N}$ image into four $\frac{\sqrt{N}}{2} \times \frac{\sqrt{N}}{2}$ quadrants and solve the problem on each quadrant recursively, so for each quadrant, each boundary pixel of that quadrant will know its nearest 1-pixel within that quadrant. Now we need to merge the quadrants so that each boundary pixel for the entire image will know its nearest 1-pixel over the entire image. Call the four quadrant UL (upper-left), UR (upper-right), LL (lower-left), and LR (lower-right). The merging takes place in two steps as illustrated in Figure PS3.1. In the first step we merge UL with UR to form U (upper) such that each boundary pixel of U will know its nearest 1-pixel over U. Likewise we merge LL and LR to form L (lower). In the second step we merge U and L to complete the process.

The actual merging of two regions is accomplished as follows: Suppose we want to merge region A and region B to form region C as illustrated in Figure PS3.2. Each pixel (u, v) on the boundary of region A knows the value $N_A(u, v)$ which is the location of its nearest 1-pixel within region A. Likewise the boundary pixels of region B have an N_B value. Now for each pixel (x, y) on the boundary of the combined region (region C) we want to compute $N_C(x, y)$. We start with those pixels (x, y) which are also on the boundary of region B. Now each such pixel (x, y) has to check each pixel (u, v) which is on the boundary of region A and adjacent to region B, and if (x, y) finds that $N_A(u, v)$ to closer than $N_B(x, y)$ then it sets $N_C(x, y)$ to be the closest such N_A value, otherwise it sets $N_C(x, y) = N_B(x, y)$. Notice

Figure PS3.1: Merging four regions into one: We start by merging UL with UR to form U and merging LL with LR to form L. Then we merge U with L to complete the process.

that there are only $O(\sqrt{N})$ such (x, y) pixels and $O(\sqrt{N})$ such (u, v) pixels so each (x, y) pixel can check each (u, v) pixel in $O(\log N)$ steps on the $\sqrt{N} \times \sqrt{N}$ MOT. Likewise the boundary pixels from region A can check against all the appropriate pixels from region B in the exact same manner in $O(\log N)$ steps.

Figure PS3.2: In merging region A with region B, each pixel (x, y) on the exterior boundary of region B (dark shade) has to check each pixel (u, v) on the interior boundary of region A (light shade) to check if $N_A(u, v)$ is closer to (x, y) than $N_B(x, y)$.

Thus we have seen that the merging can be performed in $O(\log N)$ steps on the $\sqrt{N} \times \sqrt{N}$ MOT. Thus the total running time for the first phase is

$$\begin{aligned} T_1(\sqrt{N}) &= T_1(\frac{\sqrt{N}}{2}) + O(\log N) \\ &= O(\log^2 N) \end{aligned}$$

The second phase is accomplished by working back down the recursion of the first phase, filling in nearest 1-pixel values over the whole image for the interior boundary pixels which were left behind as we created larger and larger regions in the first phase. We start with each boundary pixel of the image knowing its nearest 1-pixel over the whole image. We then fill in the nearest 1-pixel values for each of the interior boundary pixels from the four quadrants UL, UR, LL, and LR. In general at each stage we will be working in a region whose boundary pixels will know its nearest 1-pixel over the entire image. We subdivide the region into four quadrants, and we fill in the nearest 1-pixel value for each interior boundary pixel for each quadrant over the entire image. Thus we will be able to recurse into each quadrant. This situation is illustrated in Figure PS3.3.

We fill in the nearest 1-pixel value over the entire image for each interior boundary pixel of each quadrant as follows: Label the four quadrants A, B, C, and D, and consider an interior boundary pixel (x, y) which is on the boundary of quadrant A — everything works the same for the other quadrants. Due to the computation from the first phase, pixel (x, y) has the value $N_A(x, y)$ which is the nearest 1-pixel to (x, y) in region A. Each pixel on the boundary of the region (u_1, v_1) already has its value $N(u_1, v_1)$ which is its nearest 1-pixel over the entire image. Thus by checking each such (u_1, v_1), (x, y) can find out if there is a 1-pixel which might be outside the region which is closer than $N_A(x, y)$. Also each interior boundary pixel for the other quadrants (u_2, v_2) has a value which is its nearest 1-pixel within its quadrant. For example if (u_2, v_2) is a boundary pixel for quadrant D then (u_2, v_2) has the value $N_D(u_2, v_2)$. Thus by checking each such (u_2, v_2), (x, y) can find out if there is another 1-pixel from another quadrant in the region which is closer to (x, y) than $N_A(x, y)$. Thus we can compute $N(x, y)$ for each interior boundary pixel (x, y) by having each such pixel check all the exterior boundary pixels (u_1, v_1) and all the other interior boundary pixels (u_2, v_2).

If we are working in a $k \times k$ size region of the image, then there are $O(k)$ exterior boundary pixels and $O(k)$ interior boundary pixels, so we have $O(k)$ values all of which get updated by checking against $O(k)$ other values. Fortunately we have a $k \times k$ MOT over this region of the image so all the updating can be done in $O(\log k)$ steps. Thus the time to finish the second phase over the entire image is

$$\begin{aligned} T_2(\sqrt{N}) &= O(\log \sqrt{N}) + O(\log \frac{\sqrt{N}}{2}) + O(\log \frac{\sqrt{N}}{4}) + \cdots \\ &= O(\log^2 N). \end{aligned}$$

At the end of the second phase every pixel (x, y) has the value $N(x, y)$ which is its nearest 1-pixel over the entire image. Thus the total time is $T(\sqrt{N}) = T_1(\sqrt{N}) + T_2(\sqrt{N}) = O(\log^2 N)$.

Problem 5

See attached paper by: Serap A. Savari titled "Average Case Analysis of Five Two-Dimensional Bubble Sorting Algorithms" which appeared in SPAA (Symposium on Parallel Algorithms and Architectures) 1993

Problem 6

To show that any k-to-1 routing problem can be solved in $N - 1$ steps on an N-cell linear array in which each processor starts with one packet, we will use the greedy algorithm. For our model, we assume that each *edge* has a queue of size 1 associated with it. A node cannot store any packet (except at the start) that are not destined for it. Notice that each packet

Figure PS3.3: Each interior boundary pixel (light shade) (x, y) has to check each external boundary pixel (dark shade) (u_1, v_1) to see if $N(u_1, v_1)$ is closer to (x, y) than the pixel it already knows about (eg. $N_A(x, y)$), and also each other interior boundary pixel (u_2, v_2) to see if its nearest pixel (eg. $N_D(u_2, v_2)$) is closer to (x, y) than the pixel it already knows about.

moves in a single direction (towards its destination). This guarantees that in our model, packets moving in opposite directions never interfere with each other. Thus it is enough to consider packets moving in one direction only. Since each node starts out with one packet, all queues associated with edges are empty and each packet moves in its desired direction. Moreover, for every time step $t > 0$, no packet can be delayed "in transit" until it finally reaches its destination. Since the maximum distance a packet may need to move is $N - 1$ (network diameter), this time bound is sufficient to route all packets.

In our model, packets traveling in opposite directions will have no effect on each other by considering queues associated with edges only. In the worst case however, all N-packets may wish to travel in a single direction (say right). Let E_i be the set of the rightmost i nodes of the array, called the *end-zone*. Denote p_i to be the set of packets destined for E_i. Call these *priority* packets. Notice that packets in p_i have priority over other packets. Consider the rightmost priority packet that moves in the first step. This packet will never encounter any delay. Thus this packet will reach the end-zone in at most $N - i - 1$ steps. It is easy to see that the jth priority packet to reach the end-zone will do so in at most $N - i - 1 + (j - 1)$ steps. Since $j \leq \min(N, ki)$, we get the time to route all the priority packets is at most $N - i - 1 + \min(N, ki)$. This value is maximized for $i = N/k$, for a upper bound of $2N - N/k - 1$. Note that this bound is tight in the following scenario: Stack k packets in each of the left N/k processors. All N packets are destined for the rightmost N/k processors. Now, it is easy to see that N packets cross the edge $(N/k, N/k + 1)$. At this point a packet may need to travel $N - N/k - 1$ steps to get to its destination. This gives the required lower bound.

Problem 7

In the basic greedy algorithm on the $\sqrt{N} \times \sqrt{N}$ torus, each packet follows a shortest path in its sender's row to get to the correct column and then follows a shortest path in its destination column to its destination. During the row routing phase, there is never any edge contention since the packets moving in each direction move in lock step. Thus, since the diameter of the $\sqrt{N}$-node ring is $\lfloor \sqrt{N}/2 \rfloor$, each packet gets to its destination column within $\lfloor \sqrt{N}/2 \rfloor$ steps.

Now, during the column routing phase queues can build up and there may be edge contention. Edge contention is resolved by having the packet with the farthest distance remaining to travel go first – this packet is unique since no destination receives more than one packet.

To show that the column routing phase can be completed in $\lfloor \sqrt{N}/2 \rfloor$ steps, we prove the following claim by induction: After the ith step of the column routing phase, no node holds a packet which is more than distance $\lfloor \sqrt{N}/2 \rfloor - i$ away from its destination. The base case $(i = 0)$ is true since the diameter of the $\sqrt{N}$-node ring is $\lfloor \sqrt{N}/2 \rfloor$. Now suppose (for the sake of contradiction) that after the ith step some node contains a packet which is more than distance $\lfloor \sqrt{N}/2 \rfloor - i$ away from its destination. If this packet just arrived at this node at step i then one step before it must have been one edge farther away than it is now – thus after step $i-1$ it must have been more than distance $\lfloor\sqrt{N}/2\rfloor - i + 1 = \lfloor\sqrt{N}/2\rfloor - (i-1)$ away from its destination which contradicts the inductive hypothesis. On the other hand, if this packet did not move at step i then it could only have been delayed by a packet with even farther to go than this one – that is a packet which after step $i-1$ was at distance more than $\lfloor\sqrt{N}/2\rfloor - i + 1$ away from its destination which again contradicts the inductive hypothesis. Thus the claim must be true.

Finally we conclude that after the $\lfloor\sqrt{N}/2\rfloor$th step of the column routing phase, no node has a packet which is more than distance 0away from its destination. In other words, after $\lfloor\sqrt{N}/2\rfloor$ steps of the column routing phase, each packet is at its destination.

Problem 8

Consider the following permutation: Packets from the rectangle defined by the processors with indices $(0, N^{1/3}/2, 0)$, $(0, N^{1/3}, 0)$, $(N^{1/3}, N^{1/3}, 0)$ and $(N^{1/3}, N^{1/3}/2, 0)$ are destined for processors with the indices that lie in the rectangle defined by: $(N^{1/3}/2, 0, 0)$, $(N^{1/3}/2, N^{1/3}/2, 0)$, $(N^{1/3}/2, N^{1/3}/2, N^{1/3})$ and $(N^{1/3}/2, 0, N^{1/3})$ in a 1-to-1 fashion. Notice that each of these rectangles comprise of $N^{2/3}/2$ processors. Since the greedy algorithm first routes each packet along x-axis to its correct "layer" followed by routing along y-axis to the correct "column" and finally routing along the z-axis until each packet reaches its destination, for our permutation this results in $N^{2/3}/2$ packets being routed through node $(N^{1/3}/2, N^{1/3}/2, 0)$. This gives us the $\Omega(N^{2/3})$ step bound.

Problem 9

The greedy algorithm routes packets along one dimension at a time. We can analyze the total running time phase by phase.

Phase 1: Start with at most one packet per processor and route along the x-axis. It is a k-to-1 ($k \le N^{\frac{1}{3}}$) routing problem on $N^{\frac{1}{3}}$ linear array and each cell start with at most one packet. From 1.188 in problem 1, we know that $O(N^{\frac{1}{3}})$ steps suffice.

Phase 2: After phase 1, we route along y-axis. There are at most $N^{\frac{1}{3}}$ packets starting in each cell, since they must start with the same y, z coordinate. And there are at most $N^{\frac{1}{3}}$ packets going to the same destination, since they must have the same x, y coordinates in their final destinations and the original problem is one-to-one routing. Now we will use induction to show that for any greedy algorithm this phase takes at most $O(N^{\frac{2}{3}})$ steps. By symmetric, we only argue for those packets destined to its right.

Claim: 1) after $iN^{\frac{1}{3}}$ steps, any packet started in the left most i cells has either arrived its destination if its destination is in cell $k \le i$ or moved to the right of cell i if its destination is in cell $k > i$. 2) at any step, each cell contains at most $N^{\frac{1}{3}}$ packets not destined to it.

Proof: 1) $i = 0$: trivial. 2) assume true for i. For $i+1$, first prove claim 2. At each step, each cell can have at most one packet coming from left. If it previously has at least one packet not destined to it, it can have one packet going out to the right. Otherwise, it contains at most 1 packet not destined to it after this phase. Either way, it still contains at most $N^{\frac{1}{3}}$ packets not destined to it. Now for claim 1, by contradiction, first we assume that there exists a packet destined for cell $k \leq i+1$ and is not in its destination after $(i+1)N^{\frac{1}{3}}$ steps. Since we are only concerned with right-move packets, packets started at cell $i+1$ is not interesting. For packets started in cell $\leq i$, by assumption, after $iN^{\frac{1}{3}}$ steps, they should be in destination if $k \leq i$ and to the right of cell i if $k > i$, i.e., $k = i+1$ here. So this case is trivial. Now assume that there exists a packet destined for cell $k > i+1$ and after $(i+1)N^{\frac{1}{3}}$ steps, it's still in some cell $j \leq (i+1)$. By induction assumption, after $iN^{\frac{1}{3}}$ steps, this packet should have been in cell $> i$. Therefore, it did not move in the previous N steps. However, by induction, after $iN^{\frac{1}{3}}$ steps, all the packets started in cell $\leq i$ and destined for cell $> i$ has already arrived or passed cell $i+1$, i.e., no new packets will arrive at cell $i+1$ in the previous N steps, and we know from claim 2 that after $iN^{\frac{1}{3}}$ steps, cell $i+1$ contains at most N packets not destined for it, so all the packets should have moved to the right after N step. Contradiction! Thus, total $O(N^{\frac{2}{3}})$ steps will be enough.

Phase 3: Now every packet is in its right x, y coordinates, therefore, there are at most $N^{\frac{1}{3}}$ packets along each z axis, as shown in phase 2. It is a one-to-one routing problem on $N^{\frac{1}{3}}$ linear array and each cell start with at most $N^{\frac{1}{3}}$ packet. Using 1.189 in problem 1, we can finish this in $O(N^{\frac{1}{3}})$ time.

In the general case of an r-dimensional $N^{\frac{1}{r}}$-sided array, the greedy algorithm runs in r phases, routing along ith-dimension in phase i. We make the following claims:

Claim 1: After i phases, the number of packets in any one cell $\leq N^{\frac{i}{r}}$. This is easy to see by induction. Assume true for i. Then, in phase $i+1$, the most packets a cell can get are all the packets along the i dimension line, which is $\leq N^{\frac{1}{r}}N^{\frac{i}{r}} = N^{\frac{i+1}{r}}$.

Claim 2: After i phases, along dimension $i+1$, the number of packets going to the same destination $\leq N^{\frac{r-i-1}{r}}$. This is because after routing along dimension $i+1$, all the packets in a single cell should have the same $1, 2, \ldots, i+1$th coordinates.

Claim 3: Phase $i+1$ routing takes $O(\min(N^{\frac{i+1}{r}}, N^{\frac{r-i}{r}}))$ steps. From claim 1,2 and using the same induction technique, we can easily show this bound.

The overall running time of the algorithm is

$$\begin{aligned}
\sum_{i=0}^{r-1} \min(N^{\frac{i+1}{r}}, N^{\frac{r-i}{r}}) &= \sum_{i=0}^{\frac{r-1}{2}} N^{\frac{i+1}{r}} + \sum_{i=\frac{r+1}{2}}^{r-1} N^{\frac{r-i}{r}} \\
&= \begin{cases} O(N^{\frac{1}{2}}) & \text{if r is even} \\ O(N^{\frac{r+1}{2r}}) & \text{if r is odd} \end{cases}
\end{aligned}$$

Problem 10

On a 1-dimensional array with P processors we give each processor M/P items and it will take M/P steps for each processor to compute the exclusive-or of its items, and P more steps to accumulate the result in one processor. On the other hand on a two-dimensional array with P processors some processor must be given at least M/P packets to start and will thus take at least M/P steps just to read the values. Since $M \geq P^2$, $M/P \geq P$, so the 1-dimensional array will take at most twice as many steps as the two-dimensional array—i.e., they are within a constant factor of each other.

Problem 11

If $M \geq N^{s/r}$ then we can simulate the $M^{1/s}$-sided s-dimensional array on the $N^{1/r}$-sided r-dimensional array ($r < s$) with slowdown M/N as follows: First we can simulate the $M^{1/s}$-sided s-dimensional array on the $N^{1/r}$-sided s-dimensional array with slowdown $M/N^{s/r}$. Then we can simulate the $N^{1/r}$-sided s-dimensional array on the $N^{1/r}$-sided r-dimensional array with slowdown $N^{\frac{1}{r}(s-r)}$. Thus the total slowdown will be $N^{\frac{1}{r}(s-r)}M/N^{s/r} = M/N$. For smaller M we cannot use this simulation, and we are assuming that this simulation is the best possible. Thus to achieve slowdown M/N we require $M \geq N^{s/r}$.