CSE 6220 Homework 4

Karl Hiner, @khiner6

1

Consider matrix-vector multiplication $y = Ax$, where A is an $n \times n$ matrix and x and y are $n \times 1$ vectors. Output y vector elements are computed as $y[i] = \sum_{j=0}^{n} A[i,j] \cdot x[j]$. Consider matrix is partitioned with 1D column-wise partitioning, such that each processor stores $\frac{n}{p}$ complete columns of the matrix A, and has *matching* $\frac{n}{p}$ elements of the vectors x and y. I.e., if processor has column j (i.e., $A[*, j]$), it also owns $x[j]$ and $y[j]$.

Design an efficient algorithm to compute matrix-vector multiplication in parallel. Analyze the run-time of this algorithm and specify the computation and communication time.

1. Each processor P_k computes the partial sums $\hat{y_i}^k$ for its owned elements of the y vector. For each row i from 0 to $n-1$, calculate

$$\hat{y_i}^k = \sum_{j=k\frac{n}{p}}^{(k+1)\frac{n}{p}-1} A[i,j] \cdot x[j].$$

   This step takes $O(\frac{n^2}{p})$ computation time.

2. Perform an All-Reduce operation with vector addition as the operator to sum the partial sums for each y_i element:

$$y_i = \sum_{k=0}^{p-1} \hat{y_i}^k$$

   This step takes $O(n)$ computation time and $O(\tau \log p + \mu \frac{n}{p} \cdot p) = O(\tau \log p + \mu n)$ communication time.

After the All-Reduce, each processor will hold all the elements y_i of y.

Total run-time:

Computation time: $O(\frac{n^2}{p} + n)$
Communication time: $O(\tau \log p + \mu n)$

2

Given two vectors $\vec{X} = [x_0, x_1, \ldots, x_{n-1}]$ and $\vec{Y} = [y_0, y_1, \ldots, y_{n-1}]$, their convolution is defined to be the $n \times n$ matrix M whose ij^{th} entry is $M[i,j] = x_i + y_j$.

1. Give an algorithm to compute the convolution of X and Y on an n^2-processor machine. Use a logical $n \times n$ mesh topology, and assume that the vectors are initially stored in the first column of processors (one element per processor). Ignore communication constants etc. in your run-time analysis. Your algorithm should run in $O(\log n)$ time.

Observe that our goals is for each x_i to end up on all processors in the i^{th} row, and each y_i to end up on all processors in the i^{th} column. One approach would be to first broadcast each x_i across its row, then send each y_i from processor $P_{i,0}$ to the diagonal processor $P_{i,i}$, and finally broadcast each y_i across its column. This would take $O(\log n)$ time (using hypercubic permutations), and would result in no processor receiving elements it doesn't need. However, we would pay the expensive τ cost three times, and so it would be better to instead do two communication steps in the following way (at the expense of some redundant communication of y elements):

(a) Broadcast the pair (x_i, y_i) from each $P_{i,0}$ (the first column) to all processors in its row i. All processors $P_{i,j}$ with $i \neq j$ discard y_i. This takes $O(\log n)$ communication time and $O(1)$ computation time.

(b) Broadcast y_j from each $P_{j,j}$ to all processors in its column j. This also takes $O(\log n)$ communication time.

(c) On each processor, compute the sum $M[i,j] = x_i + y_j$. This takes $O(1)$ computation time.

This results in total communication time $O(\log n)$ and computation time $O(1)$.

2. Determine the computation time and communication time when using p processors by now taking the communications constants into account. What is the maximum value of p that still results in optimum efficiency? (You may assume p is a perfect square; $\sqrt{p}$ divides n etc.)

Total computation time: $O(1)$
Total communication time:

$$O(\tau \log \sqrt{p} + 2\mu \frac{n}{\sqrt{p}} \log \sqrt{p}) + O(\tau \log \sqrt{p} + \mu \log \sqrt{p} \frac{n}{\sqrt{p}}) = O(\log \sqrt{p} \left(\tau + \mu \frac{n}{\sqrt{p}} \right))$$

The serial version of this algorithm has run-time $O(n^2)$. Thus, to acheive optimum efficiency, we need

$$\log \sqrt{p} \cdot \tau + \log \sqrt{p} \cdot \mu \frac{n}{\sqrt{p}} = O\left(\frac{n^2}{p} \right).$$

The second term dominates, and we can simplify this to

$$\frac{n \log \sqrt{p}}{\sqrt{p}} = O\left(\frac{n^2}{p} \right) \implies np \log \sqrt{p} = O(n^2 \sqrt{p}) \implies p \log \sqrt{p} = O(n\sqrt{p}).$$

Thus, the maximum value of p that still results in optimum efficiency is

$$n = O(\sqrt{p} \log \sqrt{p}) \implies p = O(\frac{n^2}{\log^2 \sqrt{p}}) \implies p = O\left(\frac{n^2}{\log^2 n} \right)$$

3

Design a parallel prefix algorithm for the CRCW PRAM Sum Model, i.e., if multiple processors attempt to write to the same location, the sum of the values gets deposited there. Present an algorithm that uses $\Theta(n^2)$ processors and runs in $O(1)$ time. Argue why this algorithm is of no practical value.

We compute the (inclusive) parallel prefix sum of an input vector $\boldsymbol{x}$ of length n and store the results in output vector $\boldsymbol{y}$, also of length n, whose elements are computed with

$$y_i = \sum_{j=0}^{i} x_j, \quad i \in [0, n-1].$$

Consider a virtual arrangement of processors into n rows, where row $i \in [0, 1 \ldots, n-1]$ contains $i+1$ processors. Then we only have $\frac{n(n+1)}{2}$ processors, which is $\Theta(n^2)$.

In the CRCW PRAM Sum Model, processor (i, j), the j^{th} processor in row i can simply read the element x_j and write it to y_i, which takes $O(1)$ time.

This algorithm is of no practical value since no existing hardware, in actuality, concurrently writes to the same location from multiple processors in a single CPU cycle. Furthermore, it requires $\Theta(n^2)$ processors, which is infeasible for even modestly large values of n. (In other words, it is not work-optimal.)

4

In the mid 1980s, a team consisting of a computer scientist and a physicist at CalTech set out to build the first parallel computer. They decided to build a 64-processor configuration and use it to solve problems in physics. The physicist argued for using the three-dimensional torus interconnect as it would naturally correspond to the physics problems. The computer scientist argued for using the hypercube interconnection topology. Which choise is superior and why?

Neither choice is superior, since we can make a mapping between the two networks.

$p = 64 = 4^3$, so the 3D torus has size $m = 4$, and we need two bits to express the rank of a processor along each dimension. Thus, we need six bits to represent the rank of the processor in the hypercube:

Given mesh rank (i, j, k), the corresponding hypercube rank is $x||y||z$,

where $x = \texttt{btog}(i)$, $y = \texttt{btog}(j)$, $z = \texttt{btog}(k)$.

We can convert back from hypercube rank to mesh rank as follows:

mesh rank = $(\texttt{gtob}(x), \texttt{gtob}(y), \texttt{gtob}(z))$

This is the standard mapping between a mesh network and a hypercube network. If two processors are connected in the 3D torus, their hypercube ranks differ by only a single bit, and so they are connected in the hypercube as well.

We have shown a mapping between the two networks, which implies that any algorithm running efficiently on one network will be equally efficient on the other.

5

A pancake network of order n, denoted S_n, is defined as follows: The network has $n!$ processors. Each processor corresponds to a distinct permutation of the first n positive integers and the id of a processor is the permutation to which it corresponds. Two processors are connected iff one id can be obtained from the other by reversing the first i ($2 \leq i \leq n$) integers of its permutation. For example, 2413 is the id of a processor in S_4 and is connected to processors with id's 4213, 1423 and 3142. Show that the diameter of S_n is $O(n)$.

Let's define a `flip`(i) operation which reverses the first i integers of a permutation. Then, two processors in S_n are connected iff we can transform one processor's id into the other's id with a single flip. The distance between any two processors in S_n is then equal to the minimum number of flips required to transform one processor's id into the other's.

We want to show that the diameter of S_n is $O(n)$, so we need to show that the maximum number of flips between any two distinct permutations of the first n positive integers is $O(n)$.

Observe that, for any two processors with ids (permutations) P_1 and P_2, we can transform P_1 into P_2 by first transforming P_1 into the identity permutation (the first n positive integers, in ascending order), and then transforming from the identity permutation to P_2.

For each i from 2 to n:

1. If the integer k at position i is not at its correct position (index $k-1$), perform a `flip`(i) to move k to the first position.
2. Perform a `flip`($k-1$) to move k to its correct position.

At each iteration of the algorithm we perform at most 2 flips, for a total of $2(n-1)$ flips.

We can transform from the identity permutation to any other permutation P_2 using the same algorithm, but with the "correct" position being the position of the integer k in the permutation P_2.

Thus, we can transform $P_1 \in S_n$ into the identity permutation and then into $P_2 \in S_n$ using $2 \cdot 2(n-1) = O(n)$ flips.