CS121 Problem Set 2

Due: 23:59, November 20, 2024

1. Submit your solutions to Gradescope (www.gradescope.com).
2. In "Account Settings" in Gradescope, set FULL NAME to your Chinese name and enter your STUDENT ID.
3. If you submit handwritten solutions, write neatly and submit a clear scan.
4. When submitting your homework, be sure to match each of your solutions to the corresponding problem number.

1. Given a balanced binary tree, describe a procedure to perform all-to-all broadcast that takes time $(t_s + t_w\, m\, p\, /\, 2) \log p$ for m-word messages on p nodes. Assume that only the leaves of the tree contain nodes, and that an exchange of two m-word messages between any two nodes connected by bidirectional channels takes time $t_s + t_w\, m\, k$ if the communication channel (or a part of it) is shared by k simultaneous messages.

2. Consider the following sequential *rank sort* algorithm for n values assuming no duplicate values:

```
for (i = 0; i < n; i++) {
    x = 0;
    for (j = 0; j < n; j++)
        if (a[i] > a[j]) x++;
    b[x] = a[i];
}
```

   (a) Rewrite this as a parallel algorithm using OpenMP assuming that $p < n$ threads are used. Clearly indicate the use of private and shared variables and the schedule type.

   (b) Modify the OpenMP code in part (a) to handle duplicates in the list of values, i.e. to sort into non-decreasing order. For example, the list of values [3, 5, 7, 5, 7, 9, 2, 3, 6, 7, 8, 1] should give the sorted list [1, 2, 3, 3, 5, 5, 6, 7, 7, 7, 8, 9].

3. The following sequential code calculates a triangular matrix using a function **`calc(i,j)`**, which has no data dependences and requires a constant (but large) amount of computation.

```
for (i = 0; i < n; i++)
   for (j = 0; j <= i; j++)
   a[i,j] = calc(i,j);
```

By inserting OpenMP directives into the sequential code, show how the following schemes for assigning work to threads may be implemented on a shared memory parallel architecture, commenting on the efficiency of each scheme:

(a) A static block assignment of contiguous rows to threads.

(b) A static cyclic assignment of single rows to threads.

(c) A dynamic assignment of single rows to threads.

4. The *Back Substitution* algorithm solves a set of linear equations in upper (or lower) triangular form, as shown in Figure Q4a. A sequential algorithm to solve such a set of linear equations is given in Figure Q4b. Design a parallel algorithm for a shared memory architecture and express it using OpenMP assuming that p < n threads are used. What schedule type would you use?

$$
\begin{aligned}
a_{n-1,0}x_0 + a_{n-1,1}x_1 + a_{n-1,2}x_2 \quad \ldots \quad + a_{n-1,n-1}x_{n-1} &= b_{n-1} \\
&\vdots \\
a_{2,0}x_0 + a_{2,1}x_1 + a_{2,2}x_2 &= b_2 \\
a_{1,0}x_0 + a_{1,1}x_1 &= b_1 \\
a_{0,0}x_0 &= b_0
\end{aligned}
$$

Figure Q4a

```
/* Back Substitution */
for (i = 0; i < n; i++) {
    x[i] = b[i]/a[i][i];
    for (j = i+1; j < n; j++) {
        b[j] = b[j] - a[j][i]*x[i];
        a[j,i] = 0;
    }
}
```

Figure Q4b

5. GPU computing has been used to speed up many real-world applications. However, not all applications are suitable for GPU acceleration. Consider the following operations / applications and decide whether each is suitable for GPU acceleration or not. Briefly justify your answer. Assume all the input data are initially in the main memory.

(a) Matrix multiplications on two matrices *A* and *B*, each of size 32000 by 32000.

(b) Matrix multiplications on two matrices A and B, each of size 32 by 32.

(c) Binary search on a sorted array with 1 billion elements.

(d) Binary search on a sorted array with 1 thousand elements.

6a) Consider the following CUDA kernel for copying a matrix `idata` to another matrix `odata`. One reason for doing this is simply to test the memory bandwidth achievable on a GPU. At a high level, the kernel launches a 2D grid of thread blocks each of size `[TILE_DIM, BLOCK_ROWS]`, and each thread block copies a tile of values of size `TILE_DIM x TILE_DIM` from `idata` to `odata`. Suggested values are `TILE_DIM=32`, `BLOCK_ROWS=8`.

Give a detailed explanation of how the code works. Given an `NX x NY` matrix, how many thread blocks should be launched? Why do we want to set `BLOCK_ROWS` less than `TILE_DIM`? What does `width` represent? Why does the loop iterate over the variable `j`? What do the index calculations like `x*width+(y+j)` do?

```
__global__ void copy(float *odata, const float *idata)
{
  int x = blockIdx.x * TILE_DIM + threadIdx.x;
  int y = blockIdx.y * TILE_DIM + threadIdx.y;
  int width = gridDim.x * TILE_DIM;
  for (int j = 0; j < TILE_DIM; j+= BLOCK_ROWS)
    odata[(y+j)*width + x] = idata[(y+j)*width + x];
}
```

6b) We now modify the above kernel to transpose matrix `idata` to another matrix `odata`. Again, explain in detail how the code works. Do you expect the kernel to achieve good performance (compared to copying the matrix) when transposing a large matrix? Explain your reasoning.

```
__global__ void transposeNaive(float *odata, const float *idata)
{
  int x = blockIdx.x * TILE_DIM + threadIdx.x;
  int y = blockIdx.y * TILE_DIM + threadIdx.y;
  int width = gridDim.x * TILE_DIM;
  for (int j = 0; j < TILE_DIM; j+= BLOCK_ROWS)
    odata[x*width + (y+j)] = idata[(y+j)*width + x];
```