CSCE 5160 Parallel Processing

Exam 1: High = 80 Average = 73
Solutions posted

Problem 1:

a). Speedup = \( T_1 / T_p = (n^2) / [(n^2/p) + (n/p)^* \log p] \)
\( = (n^2 p) / [n^2 + n^* \log p] = p / [1 + (\log p) / n] \)

Efficiency = Speedup / p = 1 / [1 + (\log p) / n]

b). Iso-efficiency = \( K \cdot T_o \)
\( T_o = n^2 / p + (n/p)^* \log p \)
\( T = (n/p)^* \log p \)

iso-efficiency function = \( K \cdot (n/p)^* \log p = K \cdot (W^{1/2}/p) \log p \)

CSCE5160 March 18, 2019

CSCE 5160 Parallel Processing

c). We need set \( T_1 = T_p \)
\( n^2 = x^2/p + (x/p)^* \log p \)

Here \( x^2 \) is the new work we are trying to solve

Solving: we get \( x = -\log p + (or -) 2p^{1/2} n \) and you can further simplify this as \( O(p^{1/2} n) \)
We need \( x^2 = O(p^{*n^2}) \)

2. a). Each processor computes partial results and we need to perform reduction along ROW processors

b). Each processor receives \( (n/p^{1/2})^* (n/p^{1/2}) A \) elements and all \( n \) elements of \( B \)

Cost of all-to-one reduction is \( (t_e + t_m \cdot m) \log p \)

Since we have only \( p^{1/2} \) processors in each row and each processor has \( (n/p^{1/2}) \) involved in the reduction

Communication cost = \( (t_e + t_m \cdot ((n/p^{1/2})) \log (p^{1/2}) \)

Total cost = \( t_e \cdot (n^2/p) + (t_e + t_m \cdot ((n/p^{1/2})) \log (p^{1/2}) \)

CSCE5160 March 18, 2019
This is fairly easy. We can use either row striping or checkerboard.

<table>
<thead>
<tr>
<th>Original Rank</th>
<th>Color</th>
<th>Key</th>
<th>New Group</th>
<th>New Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

If we use row striping, each thread compares $n/p$ rows to $n/p$ columns to see if they are equal. If so, we can increment a counter (if not add a zero to the counter).

If we use checker-board, each thread compares a submatrix with another submatrix that is diagonally opposite. Again add 1 to a counter if they are equal or zero otherwise.

REVIEW

Solving Linear Equations
Gaussian elimination technique
Convert $A$ into Upper Triangular fashion

Division Step (ith row: Divide all coefficients to the right by diagonal element
Elimination Step (i+1th on, subtract a multiple of ith row to create zeros to the left of diagonal

Parallelization: Row striping
overlap communication with division step

If $p$ is smaller than $n$: use cyclical row striping

We also talked about column striping and checker-board

There will be different amounts of communication based on the partitioning
How about using column striping?

Need communication during division step but only one element is communicated

No communication for elimination step

How about using checker board striping?

Now we need communication during both division and elimination steps

OpenMP Version

```c
#pragma omp parallel private (i,j)
for (k=0; k<n-1; k++)
{
    for (j=k+1; j<n-1; j++)
    
        a[k,j] = a[k,j]/a[k,k]; # division step

    a[k,k] = 1;

#pragma omp barrier #need a barrier, no default barrier here

    for (i=k+1; i<n-1; i++)
    {
        for (j=k+1; j<n-1; j++)
            a[i,j] = a[i,j] - a[i,k]*a[k,j]; # elimination step

        b[i] = b[i] - a[i,k]*y[k];

    a[i,k]=0;
    }

#pragma omp barrier #need a barrier, no default barrier here
}
```

Schedule clause

**Static:** loop iterations are assigned at the very beginning of program execution

If chunk is specified, each thread receives that many consecutive iterations

You can specify chunk size of 1 to achieve cyclical

```c
#pragma omp parallel ... schedule(static,1)
```

If you omit chunk size, the system divides the number of iterations by # of threads

**Dynamic:** iterations are assigned either in chunks or one at a time, only when a thread completes assigned iterations

For example suppose we have 100 iterations and 5 threads. We can initially assign each thread 5 iterations. And assign additional 5 threads when a thread completes good for load balancing

**Guided:** Similar to dynamic, but the system assigns ½ of the work to threads first

Then one half of the remaining work (1/4) and so on

**Auto:** The run time system decides how to schedule iterations

We found that thread zero tends to be assigned more work

**User:** More involved but you can write your own scheduling methods
What is the fastest sequential algorithm for sorting lists?
Quick Sort -- $O(n \log n)$

This is a recursive function.

Logically, we need to divide the list of numbers to be sorted into two lists.
All numbers in one list are smaller than the numbers in the other list.

Note the two sub-lists are not sorted, only that all numbers in one list are smaller than the other. So, if we sort each list then we can do simple merge.

How do we obtain the two lists from a single list?

Find a pivot element. Start two scans of the list, one from top to bottom and another from bottom to top. The top scan continues until an element whose value is greater than the pivot is found. Likewise the bottom scan continues until an element whose value is smaller than the pivot is found. Then swap the two elements.

Stop when the two scans meet.

Pivot = 26
Two lists (19,5,15,11) (61,59,37,48,26)
Four lists (11,5,1), (15,19) (26,59,37,48), (61)
Seven lists (1,5), (11), (15), (19) (26), (59,37,48), (61)
CSCE 5160 Parallel Processing

Quicksort (list, start, end) {
    Partition(list, start, end, pivot_index); /* partitions into two lists
    Quicksort(list, start, pivot_index-1); /* sorts first list
    Quicksort(list, start, pivot_index+1); /* sorts second list
}

How do we parallelize this?
In shared memory, the list of elements are stored globally in common memory.
We can "recursively" create 2 new threads to sort sub-lists
But what is the complexity of this algorithm?

Let us assume that all lists are split into 2 equal sized sub-lists (at each recursive call)
Each thread needs to scan the sub-list
the complexity of the scan = O (size of the list)

The complexity of the parallel algorithm = n + n/2 + n/4+…2 = O(2^n) = O(n)

Compare this with sequential algorithm with a complexity of O(n log n)

Not cost optimal since the cost of parallel algorithm is O(n^2)

CSCE 5160 Parallel Processing

If we can parallelize the partitioning part and achieve O(1) complexity, then we can obtain a cost-optimal algorithm for quick sort.

Is it possible to obtain O(1) algorithm for the partitioning phases?

Let us see how we can partition lists (as per quick sort) in parallel.
The idea is to try to construct a binary tree for the purpose of partitioning.
At any stage in Quick sort, all elements smaller than pivot are on the left sub-tree and all elements greater than the pivot are on the right sub-tree.

Let us assume each processor is responsible for one element in the list.

We will assume CRCW with arbitrary write
that is if multiple processors (threads) try to write to a memory location concurrently, only one write will succeed.
CSCE 5160 Parallel Processing

(from page 403, algorithm 9.6) – **this is for shared memory only**

for (each processor $i$)

  - $root = i$;
  - $parent_i = root$;
  - processors $left[i] = right[i] = n+1$;

for (each processor $i \neq root$)

  if ($A[i] < A[parent_i]$) or
     if ($A[i] = A[parent_i]$ and $i < parent_i$)
       - $left[parent_i] = i$;
       - if $i = left[parent_i]$ exit;
       - else parent$_i = left[parent_i]$;
     else
       - $right[parent_i] = i$;
       - if $i = right[parent_i]$ exit;
       - else parent$_i = right[parent_i]$;

CSCE 5160 March 18, 2019
What is the complexity of the partitioning algorithm?

If we use n processors (as we did, with each processor being responsible for one element) the partitioning will take $O(1)$.

Note that the algorithm given does not really re-arrange the elements into two lists as the sequential version of quick sort—we simply identify pivot elements which become the roots at each level of the tree.

Since the partitioning takes $O(1)$ and the recursive portion of quick sort takes $O(\log n)$ the parallel version takes $O(\log n)$—cost optimal.

Note the tree construction results in a sorted list.

But, this is possible only if we can implement concurrent writes without using mutual exclusion (say such as read-modify in a single instruction).

We can use OpenMP’s atomic concept.

```c
for (each processor i) { 
    #pragma omp atomic 
    root = i; 
    parent, root; 
    processors left[i]= right[i]= n+1; }

for (each processor i <> root) 
    { 
        if (A[i] < A[parent[i]]) or 
            if (A[i]= A[parent[i]] and (i < parent[i]) 
                { 
                    #pragma omp atomic 
                    left[parent[i]] = i; 
                    if i = left[parent[i]], exit; 
                    else parent[i] = left[parent[i]]; 
                } 
        else 
            { 
                #pragma omp atomic 
                right[parent, ] = i; 
                if i = right[parent[i]], exit; 
                else parent[i] = right[parent[i]]; 
            } 
    }
```
Consider implementing quicksort on a hypercube (that is using MPI)

In Hypercube, we will take advantage of the property that any hypercube with \(2^d\) processors can be divided into two equal sized hypercubes with \(2^{d-1}\) nodes; and each processor in one sub-cube is connected one processor in the other sub-cube.

At each step of the quick sort, we will divide our hypercube into 2 sub-cubes.

The Algorithm works as follows.
Suppose we have \(p = 2^d\) processors and a list with \(n\) elements.
We assign \(n/p\) elements to each processor.

At each step \(i\) we select a pivot, and broadcast the pivot element to all processors.
Then all neighboring processors in dimension \(i\) (or those that differ in \(i\)-th most significant bit), exchange their elements -- one processor keeps all elements that are smaller than the pivot, and the other keeps all elements greater than the pivot element.

At the end, if we read the elements based on the processor number, we get the sorted list.

Pivot = 12
Dimension 1(most significant bit)
Now we have sorted list by reading the elements with processors according to their binary labels.

Now we have 4 sub-cubes each with two elements.

Dimension 2 (middle bit)

Dimension 3 (lsb)
CSCE 5160 Parallel Processing

Selecting a Pivot at each step is very critical to the performance. Otherwise, some processor may be idle. Consider the case where we selected the largest number as pivot in the first stage? Half of the processors will be idle after that step.

So how do we select a pivot?

If numbers are distributed uniformly, we can take the median as our pivot at each step.

How do we find a median? – can we use reduction?

In special case, if numbers are randomly distributed over a range, and if each processor is assigned \(n/p\) numbers, the median of any \(n/p\) is close to the median of all \(n\) numbers.

**Complexity?**

The algorithm performs \(d\) iterations (where \(d\) is the dimension of hypercube).

In each iteration we have

(a) pivot selection
(b) Broadcast the pivot
(c) split the numbers

Communication: broadcasting pivot

in hypercube the cost is \((t_s+t_w\times m)\log p\)

\(m = 1\) (only one element sent)

\(p\) is the number of processors – and the number changes in each iteration

\(p, p/2, \ldots\)

note \(\log p = d\), so \(\log (p/2) = d-1\), etc

Broadcast cost in step \(i = O(d-(i-1))\)

So total cost for all \(d=\log p\) steps

\[\sum_{i=1}^{d} (d-(i-1)) = O(\log^2 p)\]

Each processor must divide its list into two lists.

Let us assume that each local list will be sorted, so we have a computation cost of \(n/p\log (n/p)\) -- for sorting local lists

Then dividing the local list into two lists is simple (one unit of computation).

And then exchange these lists with its neighbor in the ith dimension (in ith iteration) involves communication
This communication costs $O(n/p)$ – since each processor has at most $n/p$ elements and all of these elements may have to be exchanged.

And since we have $d = \log p$ iterations, total communication cost for this step

$= O(n/p \cdot \log p)$

So total complexity is $O(n/p) \cdot \log(n/p) + O(n/p) \cdot \log p) + [O(\log^2 p)]$

If ignore $\log^2(p)$ – we have a cost optimal algorithm

**Merge Sort**

Let us assume each processor sorts its local numbers – takes $(n/p) \cdot \log(n/p)$

We can merge these sorted lists – we can use tree like merge (merge two lists at a time)

Merging two lists of length $(n/p)$ takes $O(n/p)$

Tree like merge takes $(n/p) + 2 \cdot (n/p) + \ldots \cdot \log(p) \cdot (n/p) = O((n/p) \cdot \log^2 (p))$

Total complexity is $O(n/p) \cdot \log(n/p) + (n/p) \cdot \log^2 (p) = O((n/p) \cdot \log(n/p))$

Here we did not consider the actual communication costs.

**Parallel Sorting by Regular Sampling (PSRS Algorithm)**

- Each process sorts its share of elements
- Each process selects $p$ regular sample of sorted list – $p$ is the number of processors
- One process gathers and sorts samples, chooses $p-1$ pivot values from sorted sample list, and broadcasts these pivot values
- Each process partitions its list into $p$ pieces, using pivot values
- Each process sends partitions to other processes
- Each process merges its partitions

Number of processors does not have to be a power of 2, unlike Hypercube.
Step 1: Each process sorts its list using quicksort.

Step 2: Each process chooses $p$ regular samples.

One process collects $p^2$ regular samples.

One process sorts $p^2$ regular samples.
One process chooses $p-1$ pivot values.

One process broadcasts $p-1$ pivot values.

Each process divides list, based on pivot values.

Each process sends partitions to correct destination process.
(first partition to $P_0$, second to $P_1$ and third to $P_2$)
Each process merges \( p \) partitions.

Complexity?

Same as the hypercube-quicksort, but we are more likely to have balanced load

We now look at another common sequential algorithm and see if we can parallelize it.