CSCE 5160 Parallel Processing

Exam 2: Wednesday April 3, 2019

HW #7. Implement Parallel Sorting by Regular Sampling (PSRS Algorithm) in MPI
Change input size n (say 100, 200 and 300)
Due March 27, 2019

Review
Sorting Algorithms
Quicksort
Quicksort on Hypercube
Merge sort

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

HW #7. Implement Parallel Sorting by Regular Sampling (PSRS Algorithm) in MPI
Change input size n (say 100, 200 and 300)
Due March 27, 2019

Step 1: Each process sorts its list using quicksort.
CSCE 5160 Parallel Processing

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

- $P_0$: 15, 54, 75, 8, 91
- $P_1$: 22, 50, 65, 5, 72
- $P_2$: 70, 83, 6, 99

**Step 3:** One process collects $p^2$ regular samples.

- 15, 54, 75, 22, 50, 65, 66, 70, 83

**Step 4:** One process sorts $p^2$ regular samples.

- 15, 22, 50, 54, 65, 66, 70, 75, 83

**Step 5:** One process chooses $p-1$ pivot values.

- 15, 22, 50, 54, 65, 66, 70, 75, 83

**Step 6:** One process broadcasts $p-1$ pivot values.
Step 7: Each process divides list, based on pivot values.

Step 8: Each process sends partitions to correct destination process. (first partition to P₀, second to P₁ and third to P₂)

Step 9: Each process merges p partitions.

Complexity?

Same as the hypercube-quick sort, but we are more likely to have balanced load.
**Simplest sorting algorithm: Bubble sort**

```plaintext
for (i = 0; i < n-1, i++)
    for (j = i +1, i<n, J++)
    {
        if (A(j) < A(i))
        {
            Temp = A(i);
            A(i) = A(j);
            A(j) = Temp;
        }
    }
```

**Sequential Complexity = \(O(n^2)\)**

**WHY?**

**ODD-EVEN Bubble sort.**

Assume that we have \(n\) processors to sort \(n\) elements. We will alternate between the ODD and EVEN phase.

In ODD phase, all **odd numbered** processors **compare & exchange** elements with their “right” neighbor (that is processor \(i\) exchanges with processor \(i+1\)).

In EVEN phase, all **even numbered** processors **compare & exchange** elements with their right neighbor (again, processor \(i\) exchanges with processor \(i+1\)).

How do we parallelize this algorithm?

We need a total of \(n\) steps (\(n/2\) odd and \(n/2\) even steps)

How much does the “compare & exchange” cost? 1 unit if no communication
Otherwise, depends on how processors can communicate.

If we assume that the \(i\) and \(i+1\) are next to each other and if we can communicate in parallel, the total communication should be \(O(n)\)

cost: for each step: 1 comparison and possibly one exchange

Total complexity = \(O(n)\) – is this cost-optimal?

What about implementing ODD-EVEN sort on a ring?

At both ODD and EVEN phase, processor \(i\) communicates with processor \(i+1\). If the ring is bi-direction, this costs only 1 for each odd or even phase.

What about if we are using a Hypercube?

How do we assure that at each step (ODD and EVEN), the processors in Hypercube are neighbors?
Gray-coding for mapping for hypercube?

Gray codes encode consecutive integers as binary where consecutive numbers differ in only one bit

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>1</td>
<td>0001</td>
</tr>
<tr>
<td>2</td>
<td>0011</td>
<td>3</td>
<td>0010</td>
</tr>
<tr>
<td>4</td>
<td>0110</td>
<td>5</td>
<td>0111</td>
</tr>
<tr>
<td>6</td>
<td>0101</td>
<td>7</td>
<td>0100</td>
</tr>
</tbody>
</table>

This way, processor numbered i will always be a neighbor of processor i+1 on the hypercube.

What about 2-D Mesh?

More difficult to assure and processor i will always be a neighbor of processor i+1.

Consider snake like numbering

What if we have \( n > p \)? Each processor is assigned \( n/p \) elements.

We start with a local sort of the \( n/p \) elements on each processor using Quick Sort \( O(n/p \log(n/p)) \)

The compare-and-exchange now requires a comparison with the two sorted lists \( O(n/p) \) comparisons.

How many ODD-EVEN steps are needed = \( p = (p/2 \text{ odd and } p/2 \text{ even}) \).

So total cost of comparisons = \( O(p^2n/p) = O(n) \)

What about communication? Each step involves \( n/p \) elements being communicated over one communication link -- \( O(n/p) \)

Since we have \( p \) steps, total cost of communication = \( O(n) \).

Total cost = \( O(\ n/p \times \log(n/p)) + O(n) + O(n) = O(n/p + \log(n/p)) + O(n) \)

Speedup = \( O(n \log n)/(O(n/p + \log(n/p)) + O(n)) \)

Efficiency = Speedup/p = \( 1/(1-O(\log p)/\log n) + O(p/\log n) \)
CSCE 5160 Parallel Processing

Efficiency = Speedup/p = 1/(1-Ω[(log p)/(log n)] + O(p/log n))

If p = O(log n)
Efficiency is approximately = 1 or the solution is cost-optimal
In other words, it is optimal for only a small workload (or n)

Note: you should be able compute iso-efficiency functions for all sorting algorithms

Consider a shared memory implementation
That is all array elements in shared memory.

We can still consider Odd – Even but now thread i (with a[i]) will compare with a[i+1] and exchange a[i] with a[i+1]

Do we need barriers?

CSCE 5160 Parallel Processing

Yes, to make sure all processors are done with “compare and exchange” in each ODD and EVEN phase

Other Sorting Algorithms

Enumeration Sort:
For each element a, find how many elements are smaller than a, so that the number can be stored in that position.

Can we can find out for any element a, its position (or rank) in the final sorted order, without performing a compare and exchange?

Consider n*n (=n²) processors to sort n elements.

In order to find the rank of one element a[i], we use n processors.
Each of these n processors will compare a[i] with one other element a[j].
Using atomic (or reduction) we “sum” on all the n processors together to count how many other elements are smaller than A[i].

Thus, with n² processors, we can obtain the rank for each of the n elements in O(1)
Once we know the rank, we store the elements in right location
Let us consider coding this algorithm

```c
/* initialize */
#pragma parallel for
for (j=0; j<n; j++)
c[j]=0; /* Note we use n threads here */

/* find the rank of a[i] */
#pragma parallel for
#pragma parallel for
for (i=0; i<n; i++)
#pragma parallel for
for (j=0; j<n; j++)
if (a[i] <= a[j])
#pragma omp atomic
    c[j]= c[j]+1; /* if atomic doesn’t work use reduce */
/* Note c[j] is the rank of a[j] */

#pragma parallel for
for (i=0; i<n; i++)
a[c[j]]=a[j]; /* Store a[i] at correct place */
```

Shear sort specifically designed for a mesh network

If we have n processors in a mesh organized as \( n^{1/2} \times n^{1/2} \) to sort n numbers.

We will use each row processors to exchange and sort their numbers

- **Even row** processors sort in descending order (smallest to the right)
- **Odd row** processors sort in ascending order (smallest to the left)

Then we sort numbers along column processors in ascending order
smallest number to the top

Repeat the phases of row sorting following by column sorting for \( \log(n) + 1 \) times
Then the final sorted list can be read off using the “snake like” numbering
Sequential complexity \((\log n + 1)^n / \sqrt{2}\)

Parallel – depends on what sorting method is used for row and column sorts

If we use quick sort and \(p = n^{1/2}\) row and column sort take \(O(\log^2 p)\) or \(O(\log^3 n^{1/2})\)

Total complexity = \(O(\log n + 1) * \log^2 n^{1/2})\)

**Bucket Sort.**

Suppose we know the total range of the values we are dealing with.

Identify \(m\) sub-ranges, called **buckets**

Place all elements into these buckets (should take no more than \(n\) time units for sequential algorithm)

Then sort each bucket. -- **We can sort each bucket using bucket sort again**
Complexity: Placing in buckets = O(n).

    Sorting each bucket = n/m log (n/m)  if we use quick sort
    = O(log(n))      if we use bucket sort recursively
    Sorting all buckets = m*[n/m log (n/m)] = n log (n/m)

Serial Bucket sort has a complexity of O(n log n/m)
If n=m than the complexity = O(n)

How to parallelize this?

Parallelization of Bucket Sort. Consider p processors. Each processor is assigned n/p elements.

    Step 1: We identify p sub-ranges for our elements (or p buckets).
    Step 2: Each processor divides its n/p elements into p buckets
    Step 3: All processors send their p buckets to appropriate processors
        At the end of this communication step, each processor has
        a single bucket to deal with
    Step 4: All processors sort their buckets -- we can use sequential version of
        bucket sort at each processor

Parallel Complexity:

Let us compute the complexity of this algorithm on Hypercube. The communication needed
for Step 3 is all-all-gather messages.

    Step 2:  O(n/p)
    Step 3:  O((n/p + (p log p))
    Step 4:  O(n/p)

Insertion Sort: Insert the number in the "right place"

use pipelined model of computation for a parallel implementation
Let us assume we have n processors numbered P_0, P_1,……P_T to sort n numbers

The data is fed to processor P_0, which keeps the smallest value so far and pass other numbers
to P_1; P_1 keeps the next smallest and sends other numbers to P_2 and so on
receive (&number, P_i-1);
    if (number < X)
        X = number;
    else send (&number, P_i+1);
CSCE 5160 Parallel Processing

We need to read the numbers from all processors: each processor can send its data to \( P_0 \), \( n-1 \) messages of size 1 – this gathering step

But some messages travel a distance of 1 while other travel a distance of \( n-1 \)

Or, \( P_{n-1} \) sends its data to \( P_{n-2} \) which sends 2 words to \( P_{n-3} \), etc.

Each message travels a distance of 1, but the size of the message increases

Communication Cost: \( 1+2+3+\ldots+n-1 = O(n^2) \)

However we can pipeline this for \( O(n) \)

Computation cost: Each processor has to compare the new number with current largest number.

Processor \( P_0 \) performs \( n-1 \) comparisons; \( P_1 \) does \( n-2 \) comparisons, etc.

Computation Cost = \( (n-1)+(n-2)+\ldots+1 = O(n^2) \)

Again these are pipelined for a complexity of \( O(n) \)

---

CSCE 5160 Parallel Processing

Sorting Networks

Consider how we used ODD-EVEN sorting. We can think of building a sorting network based on this idea. In a sorting network, we have hardware comparators.

For example we can build a simple comparator that takes 2 inputs and produces two outputs as below.

Now let us build a sorting network for 8 elements.

How many comparators are needed? We have \( n/2 \) per column and \( O(n) \) columns. So we need a total of \( O(n^2) \) Comparators

Can we come up with other ways of building sorting networks?

Time complexity? \( O(n) \)
In general we can come with any number of combinations -- people have used Genetic algorithms to find out what is the minimum number of comparisons needed.

Is it possible to think of other ways of constructing Sorting networks? Yes. Consider the Batcher’s sorting network.

Can we think of other forms of connections to form sorting networks?

Bitonic Sorting network.

Definition: A bitonic sequence is a list \(<a_0, a_1, \ldots, a_i, a_{i+1}, \ldots, a_n>\) with the property that an index \(i\) exists such that either \(a_0 < a_1 < \ldots < a_i\) and \(a_{i+1} > a_{i+2} > \ldots > a_{n-1}\), or such sub-sequences can be obtained by cyclically shifting the original list.

Examples: \((3, 5, 7, 8, 6, 4, 2)\) \(\rightarrow\) \((7, 8, 6, 4, 2, 3, 5)\) \(\rightarrow\) \((3, 5, 7, 8, 6, 4, 2)\)

If we are given a bitonic sequence, how can we create a single sorted sequence?

Consider our example: \((3,5,7)\) \((8,6,4,2)\) are the two parts of the sequence. Consider constructing two new sequences as follows.

\[ s_1: \{ \min(3, 8); \min(5, 6); \min(7, 4); \min(2)\} = \{3, 5, 4, 2\} \]
\[ s_2: \{\max(3, 8), \max(5, 6), \max(7, 4)\} = \{8, 6, 7\} \]
CSCE 5160 Parallel Processing

$s_1$: \{\min(3, 8); \min(5,6), \min(7,4), \min(2)\} = \{3, 5, 4, 2\}

$s_2$: \{\max(3, 8), \max(5,6), \max(7,4), \} = \{8, 6, 7\} - cyclic shift gives \{7, 8, 6\}

These are bitonic sequences themselves.
Also, every element of $s_1$ is smaller than every element of $s_2$.
So, now we reduced the problem of sorting the original list into sorting two smaller bitonic lists.

We can proceed this process recursively until we end up with bitonic lists of size 1.
All we have to do then is merge the lists.

Continuing with our example:

$s_{11}$: \{\min(3,4), \min(5,2)\} = \{3,2\}  \quad s_{12}$: \{\max(3,4), \max(5,2)\} = \{4,5\}

$s_{21}$: \{\min(8,7), \min(6)\} = \{7,6\}  \quad s_{22}$: \{\} = \{\}

$s_{111}$: \{2\}  \quad s_{121}$: \{3\}  \quad s_{211}$: \{6\}

$s_{112}$: \{3\}  \quad s_{122}$: \{5\}  \quad s_{212}$: \{8\}

We can now concatenate these lists to obtain sorted list.

---

CSCE 5160 Parallel Processing

So what we need is to take a list and create bitonic sequences of recursively reducing sizes. Consider the following – each node compares two numbers
Here we have a bitonic sequence with 16 elements and the elements are listed as 0000 through 1111. The values are shown.

The diagram shows a sorting network that divides the 16 element list into smaller bitonic sequences.

Time complexity? $O(\log n)$

Cost (number of comparators) = $O(n/2^k \log n) = O(n \log n)$

Each comparator can also be viewed as a processor.

So with $p = n \log n$ processors we get $T_p = O(\log n)$ and Speedup of $n$

How do we get the original bitonic sequence (in our example the original list of 16)?

We build larger and larger sequences, by creating bitonic sequences with 2, 4, 8... elements. Let us consider an example.

(8, 7, 6, 5, 4, 3, 2, 0) --- Let us look at 2 elements at a time and create bitonic sequences

Alternating ascending and descending sequences

$((7, 8, 6, 5) (3, 4, 2, 0))$ Now using these we create one sequence

$(3, 4, 7, 8, 6, 5, 2, 0)$

Again we can build a hardware unit to create bitonic sequences we use the same comparator as a node.
Let us look how to take 16 elements and construct a single bitonic sequence of 16 elements. “+” for ascending and “-” for descending.

Let us look at an example.
CSCE 5160 Parallel Processing

Let us combine the two networks (create a bitonic sequence and then sort it)

What is the sequential complexity of the combined network
Here we may consider the complexity as the depth of the network

The cost will be the total number of comparators needed
The number of columns in the last stage = log (n)
But if we now add the network to
What about the previous stage? we need log(n/2) columns

So the depth is sum of all columns and we can express this as a recurrence relationship
\[ d(n) = d(n-1) + \log(n) \]

Solving this we find \( d(n) = O(\log^2 n) \)

In the previous figure we had a total of 10 stages which is given by
\( (1+\log n)(\log n)/2 = O(\log^2 n) \)

How many comparators?
\[ n/2 \times \text{(number of columns)} = O(n/2 \times \log^2 n) \]