Parallel Algorithms

Sathish Vadhiyar

Parallel Sorting

Sathish Vadhiyar

Parallel Sorting Problem

- The input sequence of size N is distributed across P processors

- The output is such that
  - elements in each processor P_i is sorted
  - elements in P_i is greater than elements in P_{i-1} and lesser than elements in P_{i+1}

Parallel quick sort

- Naïve approach

- Start with a single processor; divide array into two sub-arrays
- Now involve one more processor
- Both the processors perform the next step of quick sort within their local subarrays
- And so on….till the number of subarrays equal the number of processors

- Disadvantage: Inefficient utilization of processors

Another algorithm

- This algorithm involves all the processors in all the iterations
- One of the processors, P0, begins by broadcasting one of its elements as the pivot element to all the processors
- Each processor then divides its local array into two sub-arrays
  - L_i: elements less than the pivot
  - G_i: elements greater than the pivot

Parallel Quick Sort

- Processors then divided into two groups:
  - First group will process the subsequent steps with L_i s
  - Second group with G_i s

- The sizes of the processor groups must be in the ratio of the number of elements in Ls and Gs to achieve load balance

- These number of elements can be found using an allreduce operation

Shared memory implementation

- All L’s are formed in the first part of the array; all G’s in the second part
- Each processor needs to know the locations in the shared memory where it has to write its L_i and G_i
- Prefix sums of the sizes of the subarrays can be used

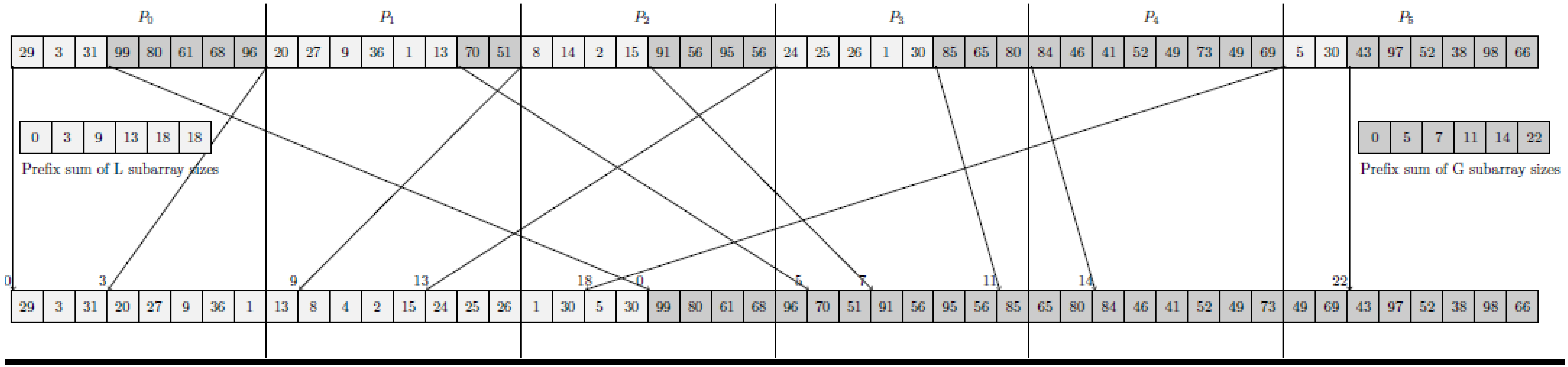
- Prefix sum can be done in O(logP)

Example: Prefix sum illustration

- In this example, 36 is the pivot element

Example: Prefix sum illustration

- In this example, 36 is the pivot element



Message Passing Version

- A processor should know which elements in its Li and Gi it should send to which processor
- Distributed prefix sum is used
- A processor can then deduce its destination processor for sending its L array using:
    - Total number of elements of L subarrays
    - prefix sums of sizes
    - Size of the processor group that will be responsible for L subarray
- Similarly for the G subarray
- In worst case, this requires all-to-all with time complexity O(N/P)

Parallel Quick sort

- The process now repeats with the subgroups
- Until the number of subgroups equal the number of processors
- At this stage, each processor performs a local quick sort: O(N/Plog(N/P))

Complexity and analysis

- log P times:
  - Broadcast: O(logP)
  - Allreduce: O(logP)
  - Prefix sum and all-to-all: O(logP + N/P)
- Then local quick sort: O(N/P.logP)

- Total: $O(N/P.\log(N/P)) + O(\log^2 P + N/P.\log P)$

- Weaknesses: Load imbalance and under-utilization

Graph Algorithms

Sathish Vadhiyar

Graph Traversal

- Graph search plays an important role in analyzing large data sets
- Relationship between data objects represented in the form of graphs
- Breadth first search used in finding shortest path or sets of paths

Parallel BFS
Level-synchronized algorithm

- Proceeds level-by-level starting with the source vertex
- Level of a vertex – its graph distance from the source
- Also, called **frontier-based** algorithm
- The parallel processes process a level, synchronize at the end of the level, before moving to the next level – Bulk Synchronous Parallelism (**BSP**) model
- How to decompose the graph (vertices, edges and adjacency matrix) among processors?

Distributed BFS with 1D Partitioning

- Each vertex and edges emanating from it are owned by one processor
- 1-D partitioning of the adjacency matrix

$$\begin{bmatrix} A_1 \\ \hline A_2 \\ \hline \vdots \\ \hline A_P \end{bmatrix}$$

- Edges emanating from vertex v is its edge list = list of vertex indices in row v of adjacency matrix *A*

1-D Partitioning

- At each level, each processor owns a set F – set of frontier vertices owned by the processor
- Edge lists of vertices in F are merged to form a set of neighboring vertices, N
- Some vertices of N owned by the same processor, while others owned by other processors
- Messages are sent to those processors to add these vertices to their frontier set for the next level

Algorithm 1 Distributed Breadth-First Expansion with 1D Partitioning

Initialize $L_{v_s}(v) = \begin{cases} 0, & v = v_s, \text{ where } v_s \text{ is a source} \\ \infty, & \text{otherwise} \end{cases}$
for $l = 0$ to ∞ **do**
  $F \leftarrow \{v \mid L_{v_s}(v) = l\}$, the set of local vertices with level l
  if $F = \emptyset$ for all processors **then**
    Terminate main loop
  end if
  $N \leftarrow$ {neighbors of vertices in F (not necessarily local)}
  for all processors q **do**
    $N_q \leftarrow$ {vertices in N owned by processor q}
    Send N_q to processor q
    Receive $\bar{N}_q$ from processor q
  end for
  $\bar{N} \leftarrow \bigcup_q \bar{N}_q$ (The $\bar{N}_q$ may overlap)
  for $v \in \bar{N}$ and $L_{v_s}(v) = \infty$ **do**
    $L_{v_s}(v) \leftarrow l + 1$
  end for
end for

$L_{vs}(v)$ – level of v, i.e, graph distance from source vs

BFS on GPUs

```
bfs_kernel(int curLevel){
v = blockIdx.x * blockDim.x + threadIdx.x;
if dist[v] == curLevel then
    forall the n ∈ neighbors(v) do
        if visited[n] == 0 then
            dist[n] = dist[v] + 1;
            visited[n] = 1;
        end
    end
end
}
```

BFS on GPUs

- One GPU thread for a vertex
- For each level, a GPU kernel is launched with the number of threads equal to the number of vertices in the graph
- Only those vertices whose assigned vertices are frontiers will become active
- Do we need atomics?
- Severe load imbalance among the threads
- Scope for improvement

Thank You