COMP 322: Fundamentals of Parallel Programming

Lecture 8: Parallel Quicksort

Vivek Sarkar
Department of Computer Science
Rice University
vsarkar@rice.edu
Announcements

• Homework 3 is due by 5pm on Monday, Feb 7th
  — This is a programming assignment with abstract performance metrics
  — To prepare for HW3, please make sure that you can compile and run the programs from Lab 2 on your own, using the -perf option. In case of problems, please send email to comp322-staff@mailman.rice.edu

• We have requested 24-hour access to Ryon building and Ryon 102 lab for all students enrolled in COMP 322

• Preferred naming convention for homework folders in clear is hw_?? e.g. hw_3
  — Please try and use this convention in the future
Acknowledgments for Today’s Lecture


• COMP 322 Lecture 8 handout


• Max Grossman for HJ code
Quicksort

• Classical sequential sorting algorithm introduced by C.A.R. Hoare in 1961 [3]

• Some reasons why Quicksort is still in use today:
  — Simple to implement
  — Worst case $O(n^2)$ execution time, but executes in $O(n \log n)$ time in practice (with high probability)
  — “In place'’ sorting algorithm -- does not need allocation of a second copy of the array.
  — Exemplar of divide-and-conquer paradigm
procedure quicksort (A, M, N); value M, N;
    array A; integer M, N;
comment Quicksort is a very fast and convenient method of sorting an array in the random-access store of a computer. The entire contents of the store may be sorted, since no extra space is required. The average number of comparisons made is 2(M – N) ln (N – M), and the average number of exchanges is one sixth this amount. Suitable refinements of this method will be desirable for its implementation on any actual computer;
begin integer I, J;
    if M < N then begin partition (A, M, N, I, J);
    quicksort (A, M, J);
    quicksort (A, I, N)
    end
end quicksort
Sequential HJ implementation of Quicksort (Listing 1)

```java
static void quicksort(int[] A, int M, int N) {
    if (M < N) {
        // partition() selects a pivot element in A[M...N]
        point p = partition(A, M, N);
        int I=p.get(0); int J=p.get(1);
        quicksort(A, M, J);
        quicksort(A, I, N);
    }
} //quicksort
```
Example Execution of Quicksort algorithm

Pivot element (can be selected randomly, or as median of three fixed elements, or by any other approach)
**Original description of partition() [2]**

**comment** I and J are output variables, and A is the array (with subscript bounds M:N) which is operated upon by this procedure. Partition takes the value X of a random element of the array A, and rearranges the values of the elements of the array in such a way that there exist integers I and J with the following properties:

- \( M \leq J < I \leq N \) provided \( M < N \)
- \( A[R] = X \) for \( M \leq R \leq J \)
- \( A[R] = X \) for \( I \leq R \leq N \)
- \( A[R] \geq X \) for \( I \leq R \leq N \)

The procedure uses an integer procedure random \((M,N)\) which chooses equiprobably a random integer \(F\) between \(M\) and \(N\), and also a procedure exchange, which exchanges the values of its two parameters;
Original code for partition() [2]
-- see Listing 1 for HJ code

```
begin  real X;  integer F;
      F := random (M,N);  X := A[F];
      I := M;  J := N;
up:   for I := 1 step 1 until N do
          if X < A [I] then go to down;
            I := N;
down: for J := N step -1 until M do
          if A[J]<X then go to change;
            J := M;
change: if I < J then begin exchange (A[I], A[J]);
                  I := I + 1; J := J - 1;
                  go to up
              end
else if I < F then begin exchange (A[I], A[F]);
                  I := I + 1
              end
else if F < J then begin exchange (A[F], A[J]);
                  J := J - 1
              end;
end  partition
```
Two Opportunities in Parallelizing Quicksort

procedure Quicksort(S) {
    if S contains at most one element then return S
    else {
        choose an element a randomly from S;
        // Opportunity: Parallelize partitioning
        let S1, S2 and S3 be the sequences of elements in S less
        than, equal to, and greater than a, respectively;
        // Opportunity: Parallelize recursive calls
        return (Quicksort(S1) followed by S2 followed by
                 Quicksort(S3))
    } // else
} // procedure
Approach 1: sequential partition, parallel calls

\[ \text{WORK}(n) = O(n \log n) \]
\[ \text{CPL}(n) = O(n) + O(n/2) + O(n/4) + \ldots = O(n) \]
Parallel HJ implementation of Quicksort for Approach 1 (Listing 2)

```java
static void quicksort(int[] A, int M, int N) {
    if (M < N) {
        // partition() selects a pivot element in A[M...N]
        point p = partition(A, M, N);
        int I = p.get(0); int J = p.get(1);
        async quicksort(A, M, J);
        async quicksort(A, I, N);
    }
} //quicksort
```
Approach 2: Parallel partition, sequential calls

\[ \text{WORK}(n) = O(n \log n) \]

\[ \text{CPL}(n) = \log(n) + 2 \log(n/2) + 4 \log(n/4) + \ldots = O(n) \]
Parallel HJ implementation of partition() for Approach 2 (Listing 3)

```java
1. static point partition(int[] A, int M, int N) {
2.   int I, J;
3.   final int pivot = M + new java.util.Random().nextInt(N-M+1);
4.   final int[] buffer = new int[N-M+1];
5.   final int[] lt = new int[N-M+1];
6.   final int[] gt = new int[N-M+1];
7.   final int[] eq = new int[N-M+1];
8.   forall(point [k] : [0:N-M]) {
12.    buffer[k] = A[M+k];
13. }
```
Parallel HJ implementation of partition() for Approach 2 (Listing 3)

14. `final int ltCount = computePrefixSums(lt);`
15. `final int eqCount = computePrefixSums(eq);`
16. `final int gtCount = computePrefixSums(gt);`
17. `forall(point [k] : [0:N-M]) {
18.     if(ltCount[k]==1) A[M+lt[k]-1] = buffer[k];
19.     else if(eqCount[k]==1) A[M+ltCount+eq[k]-1] = buffer[k];
20.     else A[M+ltCount+eqCount+gt[k]-1] = buffer[k];
21. }
22. if(M+ltCount == M) return [M+ltCount+eqCount, M+ltCount];
23. else if(M+ltCount == N) return [M+ltCount, M+ltCount-1];
24. else return [M+ltCount+eqCount, M+ltCount-1];
25.} // partition
Approach 3: parallel partition, parallel calls

\[ \text{WORK}(n) = O(n \log n) \]

\[ \text{CPL}(n) = O(\log n) + O(\log n/2) + O(\log n/4) + \ldots = O(\log^2 n) \]