Parallel Computing report

Andrea Costalonga

20/08/2021

Parallelizing Quicksort

1 Introduction

Among the algorithms used for sorting, quicksort is one of the main actors. It's performances are known to be comparable to the mergesort ones ($T(N) = O(N * log(N))$ mainly, $O(N^2)$ in the worst case) and, as we will see, also its parallel performances are not far from $T(N, P) = O(\frac{N}{P} * log(\frac{N}{P}))$.
The algorithm can be described with the following recursive formulation:

1. If the istance has at most one element, return;(base case)
2. Otherwise pick a element among the ones in the instance, that will be the PIVOT;
3. Separate the elements greater than the PIVOT (right subgroup) from the others(left subgroup);
4. Apply recursively the algorithm to the left and right subgroups;

There are various methods to pick the pivot: chosen at random, picked from the back or picking the median of a subgroup,etc. . . ; in this implementation I've gone for the second option. The usual implementation of the quicksort works *In Loco*, switching the position between the elements of the input array. In my version I've used stacks implemented via linked lists: it's a questionable choice but I really liked the challenge and the urge of doing something different from my colleagues.

2 Parallel-ify

I tried to parallelize the algorithm in the following way:

1. Partitioning the input in P partitions;(with P := # of processors)
2. Applying the sequential algorithm to those partitions;

3. Merging the partitions toghether;

Surprisingly this procedure worked flawlessly so I've gone for it. Pragmatically speaking the partitioning of the instance has been done through the method $MPI\_Scatter()$ and the merging steps using the basic methods $MPI\_Send()$ and $MPI\_Recv()$.

Analysis

For simplicity for now on we will consider N(:= instance size) a multiple of P and P a power of 2.
We can start analysing the performances of the sequential algorithm running on the $n = \frac{N}{P}$ sized partitions. We can easilly state that in the average case $T_{\text{sort}}(n) = O(n * log(n))$ since we are just applying the sequential algorithm for a smaller instance.
The parallel merger, instead, works merging in parallel couples of partitions going from P partitions to 1. With a good approximation we can assume that a good serial merging algorithm can merge two $n/2$ arrays in time $T_{\text{merge}}(n) = n$. Given the hypothesys above we can write without loss of generality the following equation:

$$T_{\text{merge}}(N, P) = \sum_{i=0}^{log_2(P)-1} (T_{\text{merge}}(\frac{N}{2^i})) \leq 2N = O(N) \tag{1}$$

Knowing that $T(N, P) = T_{\text{merge}}(N, P) + T_{\text{sort}}(\frac{N}{P})$ we can naturally derive

$$T(N, P) = O(\frac{N}{P} * log(\frac{N}{P})) + O(N) = O(\frac{N}{P} * log(\frac{N}{P})) \tag{2}$$

since $\lim_{N \to \text{inf}} \frac{N}{\frac{N}{P} * log(\frac{N}{P}))} = 0$ for any fixed P > 0.
Things work similarly for the worst case scenario, since we can just assume $T_{\text{sort}}(n) = O(n^2)$, obtaining $T(N, P) = O((\frac{N}{P})^2)$.

3 Experimental Analysis

Optimizations

Execution Times and Speedup