Martin Klampfer, 1526110 & Calvin Claus,

TU Wien | LVA: 184.710 Parallel Computing Einführung paralleles Rechnen

Fast parallel Quicksort Project

Hand-in

Index

[1. Problem statement: 2](#_Toc503293798)

[1.1. Problem definition: 2](#_Toc503293799)

[1.2. Task: 2](#_Toc503293800)

[1.3. General Notes: 2](#_Toc503293801)

[1.4. Referential, sequential implementation: 3](#_Toc503293802)

[2. Cilk 3](#_Toc503293803)

[3. OpenMP 3](#_Toc503293804)

[4. MPI 3](#_Toc503293805)

# Problem statement:

## Problem definition:

Sorting an array A of n elements by the Quicksort algorithm. The parallel algorithms should not suffer from the O(n) time partition bottleneck. In other words, find parallel approaches to doing partition in parallel.

The input is given in an array of some C basetype (int or double, …) with “<“ as the comparison function.

## Task:

Implement the specified problem in all three frameworks: OpenMP, Cilk and MPI.

## General Notes:

According to the task, we have to take a close look on the partition function because this is the time bottleneck. Our straightforward and sequential approach to the partition function looks like this:

1 void partition**(**int a**[],** int start**,** int end**,** struct partitionResult **\*** result**,**

int pivotValue**)** **{**

2 int aa**,** i**,** j**;**

3 i **=** start**-**1**;** j **=** end**+**1**;**

4

5 **for** **(;;)** **{**

6 **while** **(++**i**<**j**&&**a**[**i**]** **<** pivotValue**);**

7 **while** **(**a**[--**j**]** **>** pivotValue **&&** j**>=**start**);**

8 **if** **(**i**>=**j**)** **break;**

9 aa **=** a**[**i**];** a**[**i**]** **=** a**[**j**];** a**[**j**]** **=** aa**;**

10 **}**

11

12 result**->**smaller **=** j**-**start**+**1**;**

13 result**->**larger **=** **((**end **-** start**)** **+** 1**)** **-** result**->**smaller**;**

14**}**

In this algorithm, start and end are the starting and end indices to partition in the array a. Before the end of the method, two integer values are set in the struct partitionResult: smaller contains the number of elements smaller than the pivot value whereas larger contains the values larger than the pivot value. This method is used in all our implementations (sequential, OpenMP, Cilk and MPI) but we use the values in the struct differently. To evade the time bottleneck here, we use different approaches in the implementations. These approaches are explained further down in this document.

## Referential, sequential implementation:

As sequential reference implementation, we used a straightforward approach, which looks like this in pseudocode:

1 void quicksortS**(**int arr**[],** int low**,** int high**)** **{**

2 **if** **(**low **<** high**)** **{**

3 pivotIndex **=** get random value between low and high

4 pivotValue **=** value at position pivotIndex

5 //switch pivot to first element

6 swap**(**pivotIndex**,** low**)**

7

8 partition**(**arr**,** low**+**1**,** high**,** **&**result**,** pivotValue**);**

9 pi **=** new pivot Value index

10 swap**(**low**,** pi**)**

11

12 quicksortS**(**arr**,** low**,** pi **-** 1**);**

13 quicksortS**(**arr**,** pi **+** 1**,** high**);**

14 **}**

15**}**

//TODO Zeiten oder nicht?

# Cilk

# OpenMP

# MPI

**von seiner Angabe:**

Allowed: • Although sequential Quicksort is an in-place algorithm, your parallel solution may use extra arrays of size O(n); try to keep extra space small. Explain where and why needed.

• Optimal pivot (exact median) solution is not required. It is therefore acceptable to test on randomly generated input only (or pick pivots randomly). Introduce performance counters to measure the work imbalance (different sizes of partitioned arrays, depth of recursions)

Hint: Use MPI solution as idea also for OpenMP and Cilk; consider load balancing by prefix-sums (as in lecture)