PPAR Tutorial 1 **Master 1 in CS**

PRAM and parallel environment emulations

Getting started

We are going to work for about 20 hours on different parallelization paradigms. We will begin with simple emulations of parallel environments. We will continue in the following weeks with OpenMP, MPI and CUDA (for GPU programming). The programs you're going to write are not supposed to be submitted to your teacher unless differently specified.

You'll need to develop all your programs in C programming language. When writing the codes, you can choose and use the editor you mostly prefer.

1 PRAM Emulation

In our first exercises, we will work in an ideal parallel environment, where there is no limit on the number of processors we can employ simultaneously, and every processor can have access to the same (shared) memory, without any communication cost. The processors are supposed to execute the same list of instructions, while working on different data (SIMD machine style). Evidently, this is a theoretical setting, which will however allow us to study some interesting properties of the parallel algorithms. Synchronization issues may arise as a consequence of the fact that the memory is shared by the processors.

1.1 The sum of the first 2^k integer numbers

Our first program will have the quite simple task to compute the sum of the first n strictly positive integer numbers, where n is a power of 2, i.e. there exists $k > 0$ such that $n = 2^k$. Even if there is a simple formula for the computation of such a sum, we wish to perform an explicit term-by-term sum, for our algorithm to be suitable for any possible list of n, integer or even real, numbers.

Exercise 1.A

Implement your algorithm in C programming language, in sequential. Your implementation should:

- accept one input argument: $k \geq 0$
- ask the user the value of k when no input arguments are given
- if $k < 0$, then print, on the standard error stream, an error message
- given k, compute the value of $n = 2^k$ without using any math functions (there is a much more reliable method ...)
- use an array a of integers for storing the list of numbers to be summed (you can allocate statically the array size, but consider that the exponential grows fast as k increases)
- print the final result on the standard input stream

What is the complexity of this algorithm?

Exercise 1.B

Reasonably, the program you have developed contains a for loop which loops over the 2^k numbers to be summed. Let us now consider a theoretical environment where we can use 2^k parallel processors that will have the task to perform, each of them, one single iteration of this for loop, in parallel. Do you think your algorithm is already suitable for this kind of parallelization?

Exercise 1.C

Modify your algorithm for making it work properly in the theoretical parallel environment mentioned in Exercise 1.B. Consider the CREW (Concurrent Reading Exclusive Writing) strategy for read/write conflicts: every processor can access all elements of the array a, but only the processor i can modify the element a_i of the array, by storing partial sums. In order to implement the CREW strategy, the operations need to be performed by following the classical cascade schema over the 2^k processors. Here's an example of cascade schema for $k = 3$, which is, for $n = 8$ numbers to sum by employing $n = 8$ processors:

step 1 every processor with even rank computes $a_i + a_{i+1}$, and stores the result in a_i;

step 2 let $j = i/2$: every processor, for which j is even, executes $a_i = a_i + a_{i+2}$;

step 3 let $j = j/2$: every processor, for which j is even, executes $a_i = a_i + a_{i+4}$.

The solution will be contained only in a_0. From the example above, can you deduce a general rule (for general k) for this cascade schema?

Exercise 1.D

What is the complexity of the algorithm developed in Exercise 1.C? Give the complexity in two cases: when the algorithm is executed in sequential, and when it is executed in parallel in our theoretical environment. Recall that:

$$2^k + 2^{k-1} + 2^{k-2} + \ldots + 2^1 + 2^0 = 2^{k+1} - 1.$$

1.2 Computing the weight of tree branches

A tree $T = (V, E, w)$ is a graph, with vertex set V and edge set E, such that each of its vertices is connected, through an edge, to one and only one other vertex. Our tree T is weighted, in the sense that weights, i.e. real numbers, are associated to its edges. We are particularly interested in the weight of all tree branches starting with the tree root and ending with one of its vertices. The problem we consider consists in computing the weight of all such branches.

The code for generating a random tree and for printing its topology on the screen is available on the `share` at the following location:

`/share/m1info/PPAR/TP/TP1`

Exercise 1.E

In the code you have downloaded, there is a for loop containing no instructions. Fill the for loop with a set of instructions for computing the branch weights and for emulating a theoretical parallelism with n processors. As in Exercise 1.C, consider a CREW strategy for the read/write conflicts. The processor i is supposed to have, at the end of the execution, the weight of the branch ending at vertex i. Analyze the complexity of your algorithm in sequential and in the theoretical parallel setting.

2 Approaching the real parallelism: `fork`

Our first nontheoretical parallel implementation will be emulated with a call to the `fork` system function in C programming language. Modern multi-core CPUs can spread new threads created by `fork` over different cores of the CPU, allowing in fact to implement a primitive parallelism. Every new child thread will have a copy of the current memory of the parent.

For computing the actual execution time of your programs, we'd suggest you to use the function `gettimeofday`:

```
#include <sys/time.h>
...
int time;
struct timeval start,end;
...
gettimeofday(&start,0);
...
// your main code here
...
gettimeofday(&end,0);
time = (end.tv_sec * 1000000 + end.tv_usec) - (start.tv_sec * 1000000 + start.tv_usec);
```

2.1 Computing the weight of tree branches

We consider the same problem proposed above.

Exercise 2.A

The aim of this exercise is to separate, over 2 processors, the computational load of the for loop that you have developed for Exercise 1.E. To this purpose, we consider the `fork` system function, which is able to create a child thread from a main already-running thread. Compare the execution time to the one related to your previous implementation.

Nota bene: Only 2 processes are supposed to work here simultaneously. There is no need to exchange information between the 2 processes: at the end of the execution, they can both print their results directly on the screen.

3 To sum up ...

The complexity of the algorithms that you have developed.

Exercise 1.A:

Exercise 1.D:

sequential version	parallel version

Exercise 1.E:

sequential version	parallel version

The computational cost of your algorithms.

Exercise 1.E	Exercise 2.A