Lab Guide 6

Optimizing performance & task parallelism

Objective:

- performance measurement to optimize parallel execution
- introduce the task concept

Introduction

This lab session introduces the perf tool to profile the execution of an application code on the PU. The exercises aim to introduce performance optimizations on shared memory programming. Copy the file /share/cpar/PL06_Codigo to your home, which is a program that sorts a vector twice. Run the application with perfrecord ./a.out to sample the execution data at fixed time intervals. Use perf report to generate a [flat] view profile with the application hotspots. For better accuracy, the code (C program) should be compiled with -g -fno-omit-frame-pointer.

Exercise 1 - Critical, atomic and Reduction directives

Consider the following OpenMP program used in the previous lab session:

```
#include<omp.h>
#include<stdio.h>
double f( double a ) {
    return (4.0 / (1.0 + a*a));
}
double pi = 3.141592653589793;
int main() {
    double mypi = 0;
    int n = 10000000; // number of points to compute
    float h = 1.0 / n;
    #pragma omp parallel for reduction(+:mypi)
    for(int i=0; i<n; i++) {
        mypi = mypi + f(i*h);
    }
    mypi = mypi * h;
    printf(" pi = %.10f \n", mypi);
}</pre>
```

a) Measure the code scalability by comparing critical, atomic and reduction directives to avoid the data race in the shared variable (mypi).

To obtain the execution times of your code use <code>batch --partition=cpar time.sh</code>

b) Analyse the time overhead of the critical directive using the perf tool. Adapt the script perf1.sh for your case.

Exercise 2 - Develop an OpenMP code to implement the parallel execution of the QuickSort

```
void quickSort(float* arr, int size)
{
    int i = 0, j = size;
    /* PARTITION PART */
    partition(arr, &i, &j);
    if (0 < j) { quickSort_internel(arr, 0, j); }
    if (i< size) { quickSort_internel(arr, i, size); }
}</pre>
```

- a) Run the original code (batch --partition=cpar run.sh) and explain why one of runs takes longer to execute.
- **b)** Analyse the algorithm and suggest one parallel implementation based on omp task.
- c) Analyse the algorithm complexity of the sequential and parallel fractions, knowing that the average algorithm complexity of this sorting is Nlog₂N.
 Estimate the maximum parallelization gain for the problem size of 2048, with 2 tasks.
- d) Run the code with 2, 4 and 8 threads, measure the scalability and explain the results. Run the program with sbath --partition=cpar run2.sh. Use perf to obtain the execution profile (remove -fopenmp from the compilation and run the program with sbath --partition=cpar perf2.sh).
- e) Tuning: modify the parallelization approach to create tasks by the recursive call. Execute the program with 1, 2 and 4 threads and explain the results.
- f) Tuning: remove task creation when the size of the sub-problem is less or equal to 100 (parallelism cut-off).
 Execute the program with 1, 2 and 4 threads and explain the results.
 What is the best parallelism cut-off on this server?