Lab Guide 7

Optimizing performance & task parallelism

Objectives:

- learn how to optimize parallel execution time guided by execution profiles
- identify the basic performance limitations
- introduce the task concept

Introduction

These exercises aim to introduce performance optimizations on shared memory programming. The `perf` tool can be used to obtain the execution profile on multiple PUs, in order to identify certain kinds of bottlenecks.

The files required for this session are available on e-learning or in the cluster at `/share/cpar/PL07-Codigo`.

A [flat] view profile with the application hotspots can be obtained by running the application with `perf record ./a.out` to sample the execution data at fixed time intervals and with `perf report` to generate profile view. For better profile accuracy, the code (C program) should be compiled with `-g -fno-omit-frame-pointer`.

Exercise 1 - Overhead of `Critical`, `atomic` and `Reduction` directives

Consider the following OpenMP program from 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);
}
```

a) Compile the program and measure the [strong] scalability by comparing the sequential execution time and the parallel execution time on 2 and 4 PUs using `critical`, `atomic` and `reduction` directives to avoid the data race in the shared variable (`mypi`).
To obtain the execution times on 4 PUs use `sbatch --partition=cpar --cpus-per-task=4 time.sh`

b) Analyse the time overhead of the `critical` directive on 4 PUs 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_internal(arr, 0, j);}
        if (i< size){ quickSort_internal(arr, i, size);}
}
```

NOTE: the files for this exercise are exe2.c (which is a program that sorts a vector twice and measures the wall time and performance counters with the PAPI library), the quicksort files and the `Makefile`.

a) Build the executable and run the original code (`sbatch --partition=cpar run.sh`) and explain why one of the runs takes longer to execute.

b) Analyse the quicksort program and suggest one parallel implementation based on `omp task`.

c) Analyse the complexity of the sequential and parallel fractions of the algorithm's implementation, knowing that the average complexity of this sorting is $N\log_2 N$.
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 (note: remove the second call to the sort from the exe2.c file).
Run the program with `sbatch --partition=cpar --cpus-per-task=8 run2.sh`.

e) Tuning: modify the parallelization approach to create tasks in 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?