Lab Guide 1

Introduction to Parallelism

Objectives:

- Introduce the processor performance equation as a model that relates execution time with low level performance metrics
  - Execution time = Instructions x Cycles Per Instruction x Clock Cycle Time (Texe = #I x CPI x Tcc)
- Identify the fundamental types of parallelism: data-parallel and function-parallel
- Understand the basic parallelism limitations due to dependencies.

Introduction

The first part of this lab session uses the `perf` tool to profile the execution of the Matrix Multiplication (MM) of type double case study (both were introduced in *Lab Guide 0*[1]). The main goal is to collect performance metrics and relate these metrics with the execution time.

The students are encouraged to use their own code, developed in lab session 0, but a simple MM code can be downloaded from the course eLearning page (see the Annex 1 to use the Search cluster).

To get a better profile and more accuracy, the code (C program) should be compiled with `-g -fno-omit-frame-pointer`.

Exercise 1 - Low level metrics and the processor performance equation

a) The `perf stat` command collects a set of low level performance metrics. Run the matrix multiplication code with `perf stat ./a.out`. In the given output identify each component of the processor performance equation.

b) What is the complexity of the MM (i.e., in big O notation, where N is the problem size and NxN is the matrix size)? What increase in execution time is expected when the N doubles? Which component of the performance equation is affected (e.g., #I, CPI or Tcc)? Use the `perf stat` to compare these metrics on a 512x512 and on a 1024x1024 matrix. (note: use -r 5 to get the average of five executions).

c) Measure the gain obtained with the `O2` optimisation level on a 512x512 matrix (compared with no optimisation, i.e., `O0`? What component(s) of the performance equation is responsible for this gain?

Exercise 2 – Data parallelism and speedup analysis on a multithreaded execution

a) Develop a basic matrix multiplication using the class `std::thread` (see an example in annex B). In this exercise we will take a **data-parallel approach**: each thread will compute a set of lines of matrix C.

b) Measure the speedup of using 4, 8 and 16 threads, when compared with the base sequential execution (on a 512x512 matrix).

c) Investigate possible causes of the lack of ideal speedup in this case:

   i) By comparing low level metrics with `perf stat`

   ii) By using `perf record` & `perf report` to get a more detailed profile.

[1] The resolution of this guide assumes the resolution of the previous guide.

Exercise 3 – Function parallelism, dependency graph and intrinsic speedup limitations

The Pearson correlation coefficient is a widely used technique of compute the degree of correlation between two series of values. In this exercise we will the following non-optimised implementation to illustrate function parallelism and dependencies (see complete code on annex 3).

```
double sum(double *v, int n) {
    double sum = 0.0;
    for (int i = 0; i < n; i++) sum += v[i];
    return(sum);
}
void deviation(double *vout, double *vin,  double mean, int n) {
    for (int i = 0; i < n; i++)  vout[i] = vin[i] - mean;
}
void mdeviation(double *vout, double *vin1, double *vin2, int n) {
    for (int i = 0; i < n; i++)  vout[i] = vin1[i] * vni2[i];
}
int main() {
    …
    double avg_x = sum(x,n)/n;
    double avg_y = sum(y,n)/n;
    deviation(diff_x, x, avg_x, n);
    deviation(diff_y, y, avg_y, n);
    mdeviation(product_diff, diff_x, diff_y, n);
    mdeviation(square_diff_x, diff_x, diff_x, n);
    mdeviation(square_diff_y, diff_y, diff_y, n);
    double numerator = sum(product_diff, n);
    double sum_square_diff_x = sum(square_diff_x, n);
    double sum_square_diff_y = sum(square_diff_y, n);
    double denominator = sqrt(sum_square_diff_x * sum_square_diff_y);
    double rxy = numerator / denominator;
    printf("Pearson correlation coefficient (r): %.4f\n", rxy);
}
```

$$r_{xy} = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}}$$

a) Identify function calls that can (safely?) be executed in parallel. Draw an arrow between those functions calls.

d) Assume that all function calls take the same time to execute. What is the maximum speedup that can be obtained with this kind of parallelisation?

e) Look at the implementation of `sum`, `deviation` and `mdeviation`. What dependencies exist inside each function? How can we take advantage of that?

Annex 1: (simple) Instructions for using the search cluster:

a) **Copy local file to remote machine (don't forget the two points and point at the end) :**

```
scp <local file name> <student_id>@s7edu.di.uminho.pt:.
```

b) **Login:** `ssh <student_id>@s7edu.di.uminho.pt`

c) **Load the gcc environment:** `module load gcc/11.2.0`

d) **Compile:** `gcc -g -fno-omit-frame-pointer -O2 ...`

e) **Run:** `srun --partition=cpar perf stat -r 5 ./a.out  # assuming ./a.out on local directory`

Annex 2: Example of std::thread usage:

Compile with: `g++ -lpthread ...`

```
#include<stdio.h>
#include<stdlib.h>
#include<thread>

#define nt 2

void printid(int id) {
   printf("id=%d\n", id);
}

int main() {
    std::thread tx[nt];

    for(int i=0; i<nt; i++)
        tx[i] = std::thread(printid,i);

    for(int i=0; i<nt; i++)
        tx[i].join();

    printf("End\n");

}
```

Annex 3: Pearson correlation coefficient complete code:

```
#include <stdio.h>
#include <math.h>

#define SIZE 100
double x[SIZE], y[SIZE];


// Calculate sum
double sum(double *v, int n) {
  double sum = 0.0;
  for (int i = 0; i < n; i++) {
    sum += v[i];
  }
  return(sum);
}
// Calculate deviation
void deviation(double *vo, double *vi,  double mean, int n) {
  for (int i = 0; i < n; i++) {
    vo[i] = vi[i] - mean;
  }
}
// Multiply deviations
void mdeviation(double *vo, double *vi1, double *vi2, int n) {
    for (int i = 0; i < n; i++) {
        vo[i] = vi1[i] * vi2[i];
    }
}

int main() {
    int n=SIZE;

    double diff_x[SIZE];
    double diff_y[SIZE];
    double product_diff[SIZE];
    double square_diff_x[SIZE];
    double square_diff_y[SIZE];

       ... // init X and Y

    double sum_x = sum(x,n);
    double sum_y = sum(y,n);
    deviation(diff_x, x, sum_x/n, n);
    deviation(diff_y, y, sum_y/n, n);
    mdeviation(product_diff, diff_x, diff_y, n);
    mdeviation(square_diff_x, diff_x, diff_x, n);
    mdeviation(square_diff_y, diff_y, diff_y, n);
    double numerator = sum(product_diff, n);
    double sum_square_diff_x = sum(square_diff_x, n);
    double sum_square_diff_y = sum(square_diff_y, n);
    double denominator = sqrt(sum_square_diff_x * sum_square_diff_y);
    double r = numerator / denominator;
    printf("Pearson correlation coefficient (r): %.4f\n", r);

    return 0;
}
```