HE-IG VD | REDS Institut Reconfigurable and Embedded Digital Systems

Calcul Numérique et accélération Matérielle (CNM)

Prof Marina Zapater
Assistants : Mehdi Akeddar

Session 1 – Multithreading with OpenMP

lab03 (14.11.2024)

Objectives of the laboratory

The objectives of Lab03 are to exercise the content regarding acceleration using multithreading. Lab03 is structured in two sessions. This manual refers to "Session 1" and is devoted to understanding OpenMP basics and optimizing the performance of kNN by using OpenMP.

Laboratory validation and grading

- We will grade this laboratory based on the work we ask to be delivered.
  The parts in *light blue* are what you are expected to reply in the readme file.
- We expect (at least) three different commits to the git repository, one per session, clearly indicating the session submission (for example: git commit -m "submission session 1 lab02")
- This commit should be done at the latest the night before the next lab session.
- Therefore, you have a full week to submit the lab.

Stage 1 – OpenMP parallel execution and work sharing

OpenMP directives for C/C++ are specified with **`#pragma`** directives. The syntax of an OpenMP directive is as follows:

```
#pragma omp directive-name [clause[ [,] clause] … ] new-line
```

The **parallel** construct creates a team of OpenMP threads that execute the **parallel** region. The syntax of the **parallel** construct is as follows:

```
#pragma omp parallel [clause[ [,] clause] … ] new-line
```

Where **`clause`** is one of the following:

- **`if([parallel :]`** *`scalar-expression`***`)`**
- **`num_threads(`***`integer-expression`***`)`**
- **`default(shared ¦ none)`**
- **`private(`***`list`***`)`**
- **`firstprivate(`***`list`***`)`**
- **`shared(`***`list`***`)`**
- **`copyin(`***`list`***`)`**
- **`reduction([`***`reduction-modifier`***`,] `***`reduction-identifier`***` : `***`list`***`)`**
- **`proc_bin(master ¦ close ¦ spread)`**
- **`allocate([`***`allocator`***` :] `***`list`***`)`**

The thread that encountered the **parallel** construct becomes the master thread of the new team, with a thread number of zero for the duration of the new **parallel** region. Once the team is created, the number of threads in the team remains constant for the direction of that **parallel** region. All threads in the team, including the master thread, execute the region. Within a **parallel** region, thread numbers uniquely identify each thread. Thread numbers are consecutive numbers ranging from zero for the master thread up to one less than the number of threads in the team. A thread may obtain its own thread number within a team by a call to the **omp_get_thread_num** library routine.

The work sharing loop construct specifies that the iterations of one or more associated loops will be executed in parallel. The iterations are distributed across threads that already exist in the team that is executing the parallel region to which the work sharing loop region binds. The syntax of the loop construct is as follows:

```
#pragma omp for [clause[ [,] clause] … ] new-line
```

Where **clause** if one of the following:

- **private(***list***)**
- **firstprivate(***list***)**
- **lastprivate([** *lastprivate-modifier* **:]** *list***)**
- **linear(***list***[: *linear-step*])**
- **reduction([** reduction-modifier **,]** *reduction-identifier* **:** *list***)**
- **schedule([***modifier* **[,** *modifier***]** **:]** *kind* **[,** *chunk_size***])**
- **collapse(***n***)**
- **ordered[(***n***)]**
- **nowait**
- **allocate([***allocator* **:]** *list***)**
- **order(***concurrent***)**

The work sharing directive clauses are like the clauses of the parallel directive. One of the clauses that is worth explaining is the **schedule** clause. It specifies how iterations of the associated loops are divided into contiguous non-empty subsets called chunks and how these chunks are distributed among threads of the team.

Parallel work sharing loop construct is a shortcut for specifying a parallel construct containing a work sharing construct with one or more associated loops and nothing else.

```
#pragma omp parallel for [clause[ [,] clause] … ] new-line
```

Where **clause** can be any of the clauses accepted by the **parallel** or **for** directives, except the **nowait** clause because it is implicit within the **parallel** directive.

Other useful runtime library routines that we might need for this lab are:

- **omp_get_num_threads** returns the number of threads in the team that is executing the parallel region.
- **omp_set_num_threads** sets the number of threads to be used for subsequent parallel regions that do not specify a **num_threads** clause.
- **omp_get_wtime** returns elapsed wall clock time in seconds.

```
double start;
double end;
start = omp_get_wtime();
... work to be timed ...
end = omp_get_wtime();
printf("Work took %f seconds\n", end - start);
```

We can find much more information in the official documentation:

- OpenMP API specification (link and linked in cyberlearn).
- OpenMP API syntax reference guide (link).
- OpenMP API examples (link).

Stage 2 – OpenMP examples

The GNU Offloading and Multi-Processing Project (GOMP) is in charge to implement the OpenMP specification for the C, C++ and Fortran compilers in the GNU Compiler Collection (GCC). Different versions of the GCC might implement different versions of the OpenMP specification. If we want to use the OpenMP implementation provided by GCC we need to enable it in the compiler.

We ask you to obtain the version of the OpenMP specification is implemented in the GNU Compiler Collection suite available in our boards and how can we enable it. Answer in the README

We can use the following snippet as example for the **parallel** construct:

```c
#include <stdio.h>
#include <omp.h>

int main(void) {

    int var_1 = 1, var_2 = 2;

    // Test the effect of different data sharing clauses here
    {
        int thread_id = omp_get_thread_num();
        int num_threads = omp_get_num_threads();

        printf("Thread %02d of %02d - Vars %i, %i\n",
            thread_id, num_threads, var_1++, var_2++);
    }
    printf("Vars %i,%i\n", var_1, var_2);

    return 0;
}
```

- Execute the snippet setting the number of threads using different methods (with the OpenMP runtime routine **`omp_set_num_threads`**, the clause **`num_threads`** and the environment variable **`OMP_NUM_THREADS`**)
- Execute the snippet defining the variables *var_1* and *var_2* as **`private`**, **`firstprivate`** and **`shared`**.

We ask you to complete the README.md about the parallel construct.

We can use the following snippet as an example for the **parallel for** construct:

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

#include <omp.h>

#define SIZE 100000000

int main(void)
{
    int *vec_a = (int*)calloc(SIZE, sizeof(int));
    int *vec_b = (int*)calloc(SIZE, sizeof(int));
    int sum, i;

    double start, end;

    // Initialization, final result should be SIZE * 4
    for (i = 0; i < SIZE; i++) {
        vec_a[i] = 2;
        vec_b[i] = 2;
    }

    sum = 0;

    start = omp_get_wtime();
    // Calculate dot product
    for (i = 0; i < SIZE; i++) {
        sum += vec_a[i] * vec_b[i];
    }
    end = omp_get_wtime();

    printf("Sum %10i (%.5lfs)\n", sum, end - start);

    return 0;
}
```

- Execute this sequential snippet.
- Parallelize the dot product using OpenMP directives.

We ask you to complete the README.md about the parallel for construct.

Stage 3 – kNN

Neighbors based classification is a type of instance-based learning which means that it does not attempt to construct a general model, but it simply stores instances of the training data. Then

classification is computed from a simple majority vote of the nearest neighbors of each point. The k-nearest neighbors (k-NN) uses the k nearest neighbors to classify and k is a parameter defined by the user.

The kNN algorithm can be divided into two phases:

1. The training phase consists only of storing the training data set. It consists of vectors of multidimensional features.
2. In the classification phase, an unlabeled vector of features is classified by assigning the label which is the most frequent among the **k** training samples nearest to the unlabeled vector.

We need a distance metric to determine how far away two vectors of features are. Different metrics with different properties can be used depending on the type of data.

The optimal choice of the value k is highly data-dependent but higher values of k make the classification more robust against noise. In binary classification, it is helpful to choose k to be an odd number to avoid tied votes.

Stage 4 –Data Set

We use the Breast Cancer Wisconsin Data set. It is a multivariate data set with real features that are computed from a digitized image of a fine need aspirate (biopsy) of a breast mass. They described characteristics of the cell nuclei present in the image. You can read more about this data set in the online documentation or in the *names* file.

Ten real-valued features are computed for each cell nucleus

- Radius: mean of distances from center to points on the perimeter.
- Texture: standard deviation of gray-scale values.
- Perimeter
- Area
- Smoothness: local variation in radius lengths.
- Compactness: perimeter^2 / area - 1.
- Concavity: severity of concave portions of the contour
- Concave points: number of concave portions of the contour.
- Symmetry
- Fractal dimension: coastline approximation - 1

The data set contains a total of 569 instances with a total number of 32 attributes (the id, label and 3 values (mean, standard error and worst value) for the ten described features).

We have implemented a naïve kNN algorithm using the breast cancer Wisconsin data set. Some important aspects in the provided implementation are:

- We use the first **80%** of the samples in the file for **training** (450) and the rest for testing (119).
- Since the features in the data set are continuous real values, it uses euclidean distance as the distance metric.
- The classification result for each element in testing is the most common label in the k nearest neighbors (without weighting).

Because we use fixed elements in the data file as train and test, we introduce some randomness in the data shuffling it with the shuf command.

```
shuf data/wdbc.data > csv_wdbc_file
```

This will create a file with a random permutation of the lines in the original data set. Do not remove the shuffled file since you should use the same file during this session.

Compile the source code using the following command.

```
make
```

Inside the *Makefile* you can observe the linker flags used. We have already included the flags we need to use pthreads.

The generated program accepts two arguments, a CSV file and value of k in kNN.

```
./knn csv_wdbc_file k
```

The program will output in the terminal the time classification time, that is how long it took to compute the labels in the test set, and the confusion matrix of the prediction, where we can see how well we can diagnose breast cancer.

We ask you to measure the time to perform the classification, complete the README.md file.

Stage 5 – Implement kNN with OpenMP

We ask you to change the implementation of kNN to use OpenMP. You must indicate which part of the algorithm you parallelized and why.

We ask you to implement kNN with OpenMP parallelization, measure the time it takes to perform the classification and complete the README.md.

Before you leave…

Make sure you have all the requests *in light blue* of this manual, that you have implemented all the functions required and filled-in the answers in the *README* file in the repo. Then commit all files to your own “cnm24-yourname” repository.

We expect (at least) one commit for this session, clearly indicating the session submission (for example: git commit -m “submission session 1 lab03”)

This commit should be done at the latest the day before the next lab session at 23:59h.