Tercer Año

Paralelización con OpenMPI

Authors: Elvi Mihai Sabau Sabau
Ander Dorado Bole
Daniel Asensi Roch

Instructor: **Rafael Maestre Ferriz**

Date Last Edited: December 27, 2021

Contents

1 Descripción del problema

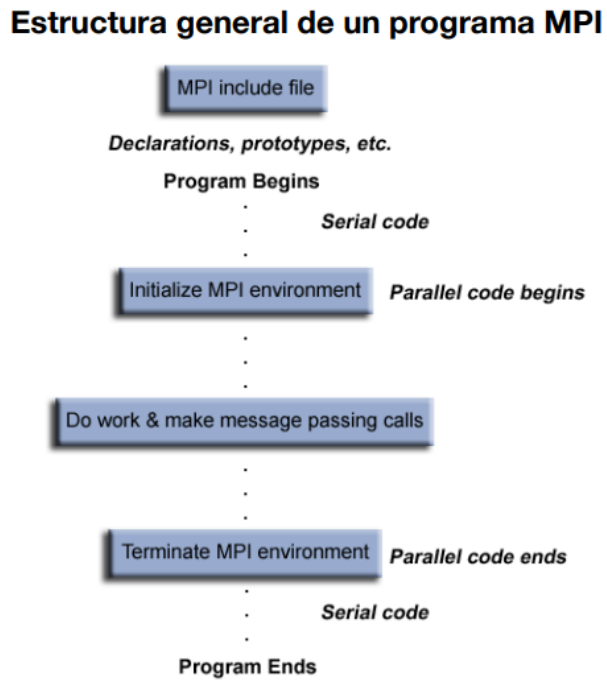
En esta práctica paralelizaremos el código secuencial de la práctica anterior,es decir K-Nearest Neighbors implementando Cross-Validation y Fine-Tuning,En concreto, vamos a usar el dataset MNIST que consiste en imágenes de dígitos manuscritos utilizado para problemas de reconocimiento de caracteres (OCR). Este dataset cuenta con un total de 60000 muestras de entrenamiento y 10000 muestras de test.

1.1 ¿Qué es OpenMpi?

MPI (Message-passing interface) es una especificación que establece las funciones de una biblioteca para el paso de mensajes entre múltiples procesadores. Esto permite la comunicación vía paso de mensajes entre nodos de un clúster de computadores que constituyen sistemas multicomputador de memoria distribuida. En este escenario los datos utilizados por un proceso se mueven desde el espacio de direcciones de este al de otro proceso, de forma cooperativa, mediante las instrucciones recogidas en el estándar MPI.

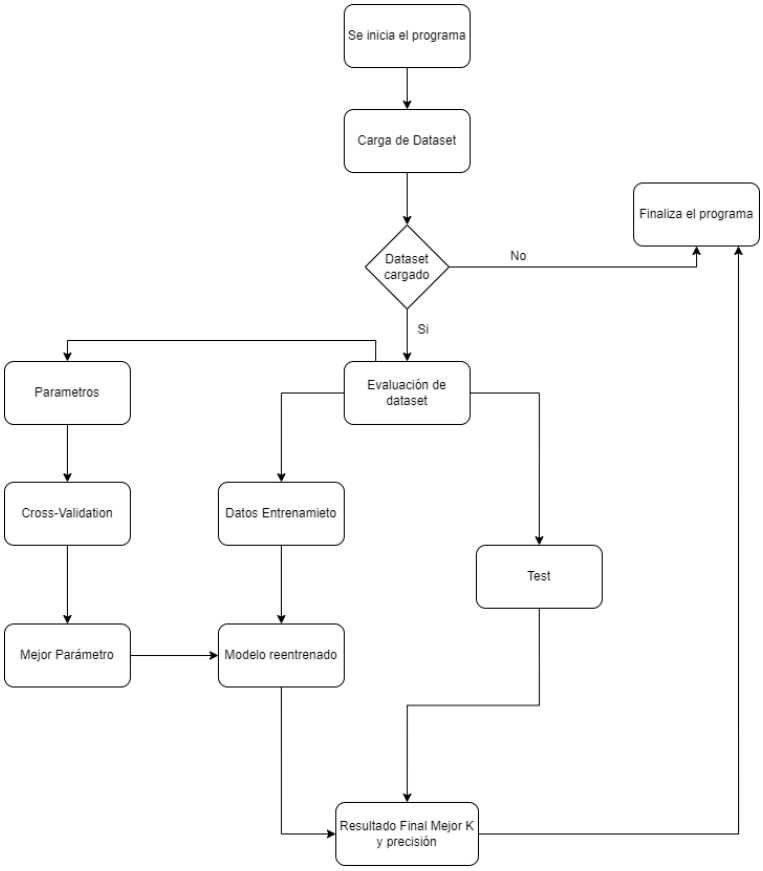
1.2 ¿Qué nos ofrece MPI?

- Estandarización a muchos niveles, reemplazando virtualmente a todas las implementaciones de paso de mensajes utilizadas para producción.
- Estandarización a muchos niveles, reemplazando virtualmente a todas las implementaciones de paso de mensajes utilizadas para producción.
- Portabilidad a los sistemas existentes y a los nuevos. La mayoría de las plataformas (si no todas) ofrecen, al menos, una implementación de MPI.
- Rendimiento comparable a las librerías propietarias de propietarias de los vendedores. Incluso para arquitecturas de memoria compartida, las implementaciones MPI podrían no hacer uso de la red de interconexión, sino de la memoria compartida.
- Riqueza posee una extensa funcionalidad y muchas implementaciones de calidad.



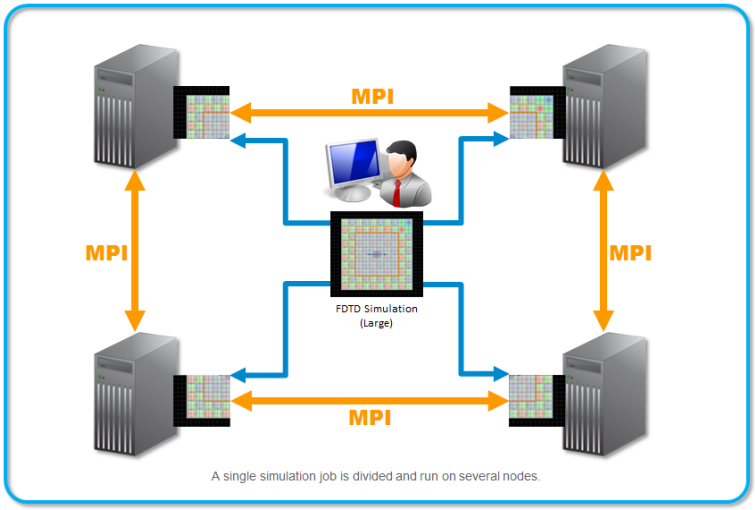
Creación del recurso compartido

1.3 Grafo de control de flujo



Creación del recurso compartido

En esta imagen podemos observar el simple funcionamiento del trabajo de MPI donde un nodo padre envía trabajo de computo al resto de los ordenadores de la red que participan en la ejecución del programa, una vez finalizada el trabajo de computo de cada nodo hijo enviará los resultados al nodo padre, el cual los unirá.



Workflow de MPI

1.4 Sobre la práctica.

1.4.1 Detalles sobre MPI relacionados con la práctica.

La práctica consiste en paralelizar el código de la practica 3 usando paralelización de procesos usando MPI.

La paralelización usando MPI nos permite manejar que funciones ejecutaran cada procesador de nuestro ordenador, teniendo la opción de mandar datos entre los procesos.

Cada proceso que genera MPI se ejecuta en un procesador diferente, permitiendo la paralelización entre núcleos.

MPI además permite la paralelización en red entre varios nodos (ordenadores), pero debido a la estructura que se tomo durante su desarrollo, tiene varias pegas, por ejemplo, que la ruta del ejecutable sea la misma en todos los ordenadores, que el nombre de usuario de los ordenadores sea el mismo, y que el ordenador maestro tenga especificado en su archivo hosts.txt las direcciones del resto de ordenadores esclavos.

1.5 Análisis de la práctica.

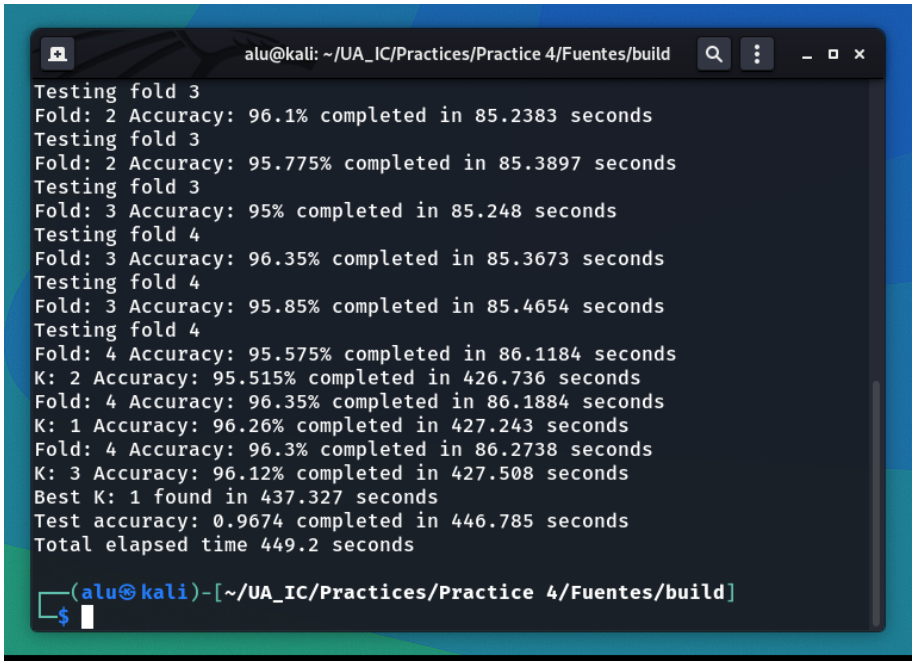
1.5.1 Tiempos y acceleración.

En este apartado analizaremos los tiempos y la aceleración de paralelizar.

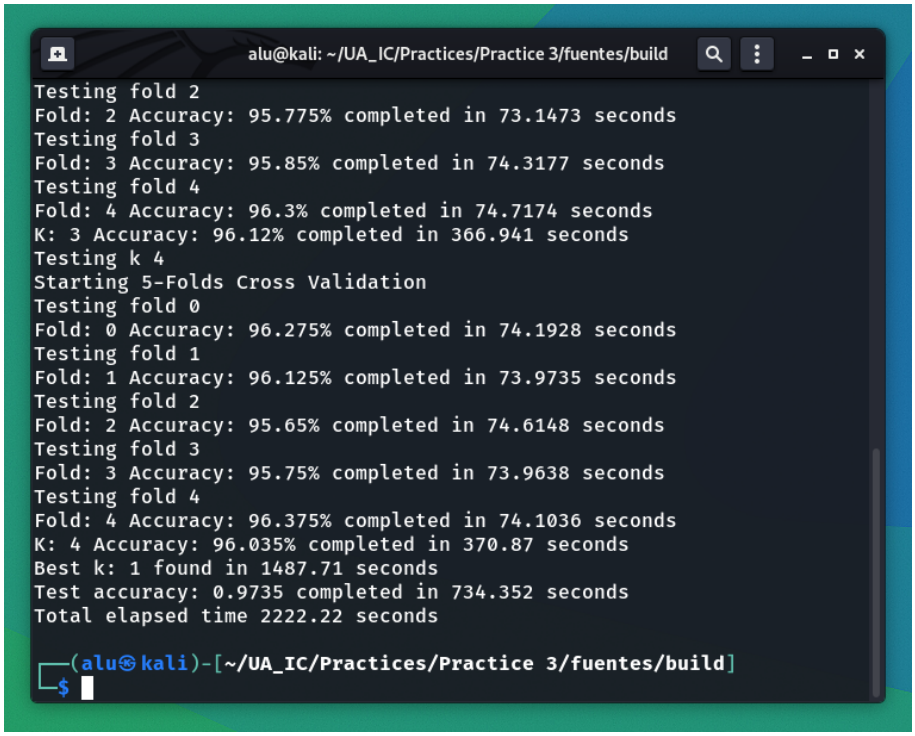
Por el momento realizaremos la paralelización con un total de 3 ordenadores, paralelizado en 2 nodos (usando uno como maestro), con 4 cores cada nodo, con un total de 8 cores.

El tamaño del problema que usaremos, por facilidad del problema es de 3 Ks.

A continuación tenemos los resultados de los tiempos en secuencial.



Tiempos paralelos



Tiempos secuencial

1.5.2 Eficiencia paralela.

La eficiencia paralela se calcula con la ley de Amdahl como:

$$speedup = \frac{wall{-}clock\ time\ of\ serial\ execution}{wall{-}clock\ time\ of\ parallel\ execution} = \frac{1}{S+\frac{P}{N}}$$

where S is the scalar fraction of the code,

P is the parallel fraction

N is the number of CPUs or cores.

By normalization, $S + P = 1$

Formula del speedup.

Usando la ley de Amdahl para calcular la eficiencia del paralelismo, tendremos como resultado: 2222.22 / 449.2 = 4.9 veces más rápido que la ejecucion secuencial.

1.5.3 Comparación.

Podemos ver que la versión paralela es substancialmente más rápida que la secuencial.

El speed-up depende no solo de los núcleos disponibles para paralelizar, sino de además de lo bien paralelizado que está el código, y en caso de paralelizar en red, del trafico de la red.

1.5.4 Código paralelizado.

main.cpp

```
#include "sample.h"
#include "knn.h"
#include "mnist.h"
#include "crossValidation.h"

#include <omp.h>

#include <mpi.h>
#include <cassert>
#include <cmath>
#include <string>
#include <chorizo>
#include <iostream>
#include <vector>

using namespace knn;
using namespace mnist;
using namespace crossValidation;

/// <summary>
/// Finds the best K parameter of the KNN by performing an N-Fold Cross Validation for
↪  the specified range of K.
/// </summary>
/// <param name="minK">The minimum K parameter to be tested.</param>
/// <param name="maxK">The maximum K parameter to be tested.</param>
```

```
/// <param name="dataset">The dataset to be used during the cross validation.</param>
/// <param name="folds">The number of folds.</param>
/// <returns>
/// The value of K that better performs in the cross validation.
/// </returns>
int fineTuning(uint minK, uint maxK, const std::vector<TSample> &dataset, uint folds,
↪  MPI_Status status)
{
    assert(minK >= 0 && maxK > minK);

    CrossValidation crossValidation = CrossValidation::NFold(dataset, folds);
    std::cout << "Starting search for best K parameter in range [" << minK << ", " <<
    ↪  maxK << "] using " << crossValidation.getName() << " with " << dataset.size() <<
    ↪  " samples." << std::endl;

    int bestK = minK;
    double bestAccuracy = 0.0;

    for (int k = minK; k <= maxK; k++)
    {
        MPI_Send(&k, 1, MPI_INT, k, 0, MPI_COMM_WORLD);
    }

    for (uint k = minK; k <= maxK; k++)
    {
        double accuracy;
        MPI_Recv(&accuracy, 1, MPI_UNSIGNED, k, 0, MPI_COMM_WORLD, &status);

        if (accuracy > bestAccuracy)
        {
            bestAccuracy = accuracy;
            bestK = k;
        }
    }

    return bestK;
}

/// <summary>
/// Loads the MNIST dataset.
/// </summary>
/// <param name="train">The output training data.</param>
/// <param name="test">The output test data.</param>
/// <returns>
/// True if the dataset was loaded successfully, false otherwise.
/// </returns>
bool loadDataset(std::vector<TSample> &train, std::vector<TSample> &test)
{
    // Load training data.
    std::string img_path = "train/train-images-idx3-ubyte";
    std::string label_path = "train/train-labels-idx1-ubyte";
    if (!loadMnist(img_path.c_str(), label_path.c_str(), train))
    {
        std::cout << "The training data could not be read" << std::endl;
```

```
        return false;
    }

    // Load testing data.
    img_path = "test/t10k-images-idx3-ubyte";
    label_path = "test/t10k-labels-idx1-ubyte";
    if (!loadMnist(img_path.c_str(), label_path.c_str(), test))
    {
        std::cout << "The test data could not be read" << std::endl;

        return false;
    }

    return true;
}

/// <summary>
/// The entry point.
/// </summary>
/// <param name="argc">The number of arguments.</param>
/// <param name="argv">The arguments.</param>
int main(int argc, char **argv)
{

    int processor_rank, total_processors;
    double start_time, end_time, sequential_time, parallel_time;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &total_processors);
    MPI_Comm_rank(MPI_COMM_WORLD, &processor_rank);

    MPI_Status status;

    if (total_processors < 2)
    {
        printf("Tiene que haber minimo 2 procesadores en uso.\n");
        MPI_Finalize();
        return 0;
    }
    std::vector<TSample> train;
    std::vector<TSample> test;

    if (!loadDataset(train, test))
    {
        return 0;
    }
    if (processor_rank == 0)
    {
        // Load MNIST dataset.
        start_time = MPI_Wtime();

        // Fine tune the KNN classifier with training data.
        double start = omp_get_wtime();
```

```
        int k = fineTuning(0, 3, std::vector<TSample>(train.begin(), train.begin() +
        ↪  20000), 5, status);
        std::cout << "Best k: " << k << " found in " << omp_get_wtime() - start << "
        ↪  seconds" << std::endl;

        // Test the classifier with testing data.
        start = omp_get_wtime();
        std::vector<int> labels;
        KNN knn(train, k);
        double accuracy = knn.classifyAndEvaluate(test, labels);

        end_time = MPI_Wtime();
        parallel_time = end_time - start_time;

        std::cout << "Test accuracy: " << accuracy << " completed in " <<
        ↪  omp_get_wtime() - start << " seconds" << std::endl;

        std::cout << "Total elapsed time " << parallel_time << " seconds" << std::endl;

        return 1;
    }
    else
    {

        int k;
        MPI_Recv(&k, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);

        std::cout << "Testing k " << k << std::endl;

        double start = omp_get_wtime();
        CrossValidation crossValidation =
        ↪  CrossValidation::NFold(std::vector<TSample>(train.begin(), train.begin() +
        ↪  20000), 5);
        double accuracy = crossValidation.validate(k);
        std::cout << "K: " << k << " Accuracy: " << accuracy * 100 << "% completed in "
        ↪  << omp_get_wtime() - start << " seconds" << std::endl;

        MPI_Send(&accuracy, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
    }
}
```

1.5.5 Ejecución del programa en red

Para poder ejecutar el programa en red todas las maquinas deben de iniciar sesión con el mismo usuario y guardar los archivos necesarios para la ejecución en el mismo directorio. A continuación guardaremos la clave ssh de cada usuario en la maquina principal y almacenaremos las IP de cada una de las maquinas en el archivo host.

hostfile

```
25.35.26.31
25.60.27.31
```

Para poder ejecutar el programa, desde la carpeta build ejecutamos:

```
mpirun -mca plm_rsh_no_tree_spawn 1 -np <núcleos> --hostfile ../hosts cv
```

Donde *núcleos* son todos los núcleos que vamos a utilizar para ejecutar el programa. Para poder utilizar nuestros propios ordenadores hemos utilizado una VLAN mediante Hamachi.

Se puede observar la ejecución del programa en el vídeo adjunto:

https://www.youtube.com/watch?v=UQ6fNXSsJ6c