Pontificia Universidad JAVERIANA Cali

Facultad de Ingeniería y Ciencias
Carrera de Ingeniería de
Sistemas y Computación

# PROGRAMACIÓN PARALELA
# Intermediate MPI Tutorial

In the accompanying archive, there is a compressed directory called `Lab2`. Once the archive is uncompressed change to that directory. Inside it there is a `Makefile`. Use `make` to compile the programs in that directory.

The first program you should execute is called `probe-status`. The program illustrates how to receive a message whose exact size is unknown at compile time. For that purpose, `MPI_Probe`, the `MPI_Status` C structure, and `MPI_Get_Count` can be used to determine the incoming message size and then, dynamically allocate a buffer big enough before invoking `MPI_Recv`. The program is meant to be executed by two processes.

The next program to test is called `cpi-serial`. The program estimates the value of pi by means of the integral of `f (x) = 4 / ( 1 + x^2 )` using the tangent-trapezoidal rule using `n` trapezoids. It employs a sequential approach to do so. The `f (x)` function is not written as a C function, since doing it that way will incur in an unnecessary overhead in each loop iteration. Instead, it is written as a function-like macro, which preserves code legibility and does not incurs in any overhead. The number of trapezoids is entered by the user. The program executes inside an endless loop which is broken when the user enters `0`.

The next program to test is called `cpi_p2p`. This is the first parallel version; it makes use of blocking, point-to-point message primitives (MPI_Send/MPI_Recv) to send/receive messages. The root process sends a first message, denoted with the `N_TAG` tag, corresponds to the number of trapezoids; this message is sent to all the processes in the job. The purpose of this message is to partition the task among all the processes. Once every process has evaluated the sum of the areas of its corresponding trapezoids, it sends this sum (`my_pi`) to the root process. Once the root process has received all contributions, the `pi` variable holds the accumulated result.

The next program to test is called `cpi_copns`. This second parallel version makes use of two widely used collective operations: `MPI_Bcast` and `MPI_Reduce`. The former one is used to send the same piece of data from a given root process to all processes in a communicator. The triple (`&n`, `1`, `MPI_INT`) indicate that the process with rank `0`, will send the number of trapezoids to all the processes in the `MPI_COMM_WORLD`. Processes with a rank different to `0`, will receive the number of trapezoids in its own address space via the `n` variable. Every process computes its own contribution in the my_pi variable. To consolidate the contributions, `MPI_Reduce` is used: the process with rank `0` receives the contributions and adds them (according to the `MPI_SUM` operator) in the `pi` variable; the remaining processes send the contributions to the process with rank `0` in the `my_pi` variable.

The next program to test is called `MPITimeIt0`. The program illustrates how to measure execution times using `MPI_Wtime`. The workload is simulated by calling the `sleep` function, which will block the invoking process for the given duration in seconds.

The next program to test is called `MPITimeIt1`. This program simulates different workloads for each process, and thus is a better approach than the previous one. This is due to a lack of global synchronisation in the nodes in the parallel machine, as well as not being able to balance evenly the workload among all the processes.

The next program to test is called `MPITimeIt2`. This program uses `MPI_Reduce` to compute the minimum, maximum, and average execution times.

The next program to test is called `MPITimeIt3`. It uses `MPI_Reduce` to synchronise the starting time of all the processes. The execution time in each node is sent to the root process, which prints the execution times.

The next program to test is called `MPITimeIt4`. This is a variant of the previous program; it uses `MPI_Gather` to collect the execution times.

The next program to test is called `scatter-gather`. Assume we need to compute the average across all numbers in an array using a parallel approach. The entire array is located in the root process and its size is unknown at compile time; therefore we will have to allocate memory in a dynamic manner. When the program is invoked we need to provide the number of elements to store in each process as an argument in the command line. Each element of the array will have a predetermined content: its index plus one. We use `MPI_Scatter` to split the entire array among the processes. Each process computes the average of its elements. The root process allocates memory to store the local averages and collects them using `MPI_Gather`; then it computes the global average. Before exiting, we need to return the allocated memory to the heap.

The next program to test is called `scatter-allgather`. This is a variant of the previous program. The difference is that now we need the global average in every process. Hence, we will use `MPI_Allgather` to distribute the local average so that each process can compute the global average.

The last program to test is called `allreduce`. Assume we need to compute the average and the standard deviation across all numbers in an array. The array can be split among all of the processes as before. Then, we locally compute the sum of all elements in the subarray. Since we need the mean in each process, we use `MPI_Allreduce` to compute the global sum and have a local copy in every process to obtain the mean. Then, we compute the local sum of the squared differences from the mean. Afterwards, we use `MPI_Reduce` to compute the global sum of the squared differences on the root process; that is, the variance. Finally, the root process compute the standard deviation and prints it as well as the mean.