Advanced Parallel Programming

Labs2

MPI Non-Blockings Communications

Michel Herrera Sanchez

1. Parallelization of pi\_integral.c using non-blocking communication

The parallelization of code pi\_integral.c mainly consists on split the intervals for the integral calculation equally among a set of process. The number of iterations is initially sent to each process using the function MPI\_Ibcast from process with rank zero. Every MPI-Non blocking communication requires a matching MPI\_Wait that’s is called afterwards. Then the calculation of the integral is done on each process after the number of iteration is known. Afterwards the results of all partial calculations per process need to be globally reduced using a SUM operator and stored in process with rank 0. The global reduce operation was implemented calling the MPI function MPI\_Ireduce. At the end of the program the process with rank 0 shows the results.

For the current experiment 16 mpi process were involved in PI calculation with 1000000 iteration.

As shown in the following table non-blocking version of p\_integral parallelization is slighter slower than the pure implementation.

|  |  |
| --- | --- |
| Implementation | Execution Time (s) |
| MPI | 0.003517 |
| MPI Non-Blockings Communications | 0.004009 |

See code pi\_integral\_mpi\_acoll.c for implementation details.

1. Parallelization the code dotprod.c using MPI nonblocking collectives.

The current solution assumes that the number of elements of each dotprod vector is divisible by the number of mpi process. The number of vector’s elements is initially sent to each process using the function MPI\_Ibcast from process with rank zero. Data is initially loaded in process with rank zero for the two operands vectors x and y. Then, two MPI\_Iscatter calls are performed using root process the one with rank 0. The first call to MPI\_Iscatter distributes equals chunk sizes of portions of vector X among all process. The same happen with the second MPI\_Iscatter call that distributes equally chunk sizes of vector Y among all the process. Due to the mature of asynchronous MPI calls an MPI\_Wait is called to match a non-blocking MPI call. Given the fact that it’s an asynchronous scatter operation none of the process need to wait for the completion of the scatter operation globally. Instead, right after receiving the X and Y chunks each process perform the dotprod multiplication with the portion of the vectors placed in his local memory. The process with rank 0 lead a global reduction operation calling method MPI\_Ireduce to collect and sum all the partial dotprod results present on each of the process. Once it’s done the process with rank 0 print out the results.

As shown in the following table the non-blocking version of dotprod shows a better performance than the pure mpi blocking solution. For the current experiment 16 mpi process were involved in the distributed dotprod calculation with vectors of 16384 elements.

|  |  |
| --- | --- |
| Implementation | Execution Time (s) |
| MPI | 0.00105 |
| MPI Non-Blockings Communications | 0.000776 |

See code dot\_prod\_mpi\_acoll.c for details.

1. Parallelization the code mxvnm.c using MPI Nonblocking collectives.

The current solution assumes that the number of rows/columns of the matrix and equals to the number of elements of the vector. The number of columns of the matrix should be divisible by the number of mpi process.

Before partial matrix/vector multiplication happens on each of the participant MPI process. The number of rows and columns of the matrix are broadcast to all process using the MPI function MPI\_Bcast. Then each process receives a submatrix from the original matrix with the same number of columns but with a number of rows equals to number of rows of the matrix divided by the number of process. This was relatively easy to implement using the MPI\_Iscatter function. A problem arises to send the same copy of vector X to all the process due to broadcast was not a suitable method. To send exactly the same copy of vector X to all process, we took the opposite approach. Each process initialize an equal size portion of the vector and then using the call to MPI\_Iallgather it was possible to copy to each process exactly the whole vecto X with all chunks coming from the different process. What follows is the partial multiplication of each submatrix privately stored on each process with the whole vector X. At the end process with rank 0 got the partial results from each process to fill the rows for the vector Y and final result. Each process produce one element of vector Y and it’s finally collected in process with rank 0.

As shown in the following table the pure mpi version of mxvn shows a better performance than the pure mpi non-blocking solution. For the current experiment 16 mpi process were involved in the distributed multiplication of a matrix with rows/columns8192x8192 and a vector of 8192 elements.

|  |  |
| --- | --- |
| Implementation | Execution Time (s) |
| MPI | 0.087563 |
| MPI Non-Blockings Communications | 0.258770 |

See code mxvnm\_mpi\_acoll.c for details.