Advanced Parallel Programming

Labs 3

MPI: Remote Memory Access Operations

Michel Herrera Sanchez

1. Parallelization of pi\_integral.c using RMA operations

The parallelization of pi\_integral using RMA operations is closed to the mpi implementation except that all communication is done by using RMA Windows. The number of iterations is published by process rank 0 in RMA window nwin and read from there by all process using the RMA function MPI\_Get on RMA window nwin. After that each process start the calculation of the integral on each interval. Once calculation is finished the partial results are stored and reduced in the second RMA Window piwin using the method MPI\_Accumulate and let the global value of pi on target process with rank 0.

For the current experiment 16 mpi process were involved in the benchmark of pi calculation with 1000000 iterations.

As shown in the following table mpi rma version of pi\_integal parallelization is slower than the pure MPI implementation.

|  |  |
| --- | --- |
| Implementation | Execution Time (s) |
| MPI | 0.0036622 |
| MPI RMA | 0.004541 |

See code pi\_integral\_rma.c and p\_integral\_mpi.c for implementation details.

2. Parallelization of dotprod.c using RMA operations

The parallelization of dotprod using RMA operations is not too different from the pure mpi implementation. Nevertheless, all communications are performed using RMA operations. Two RMA Windows were created xwin, ywin, dotwin. The first two windows stores the values of vector X (xwin) and vector Y (ywin). Process with rank 0 uses those RMA windows to provide access to those vector across all process. Except that each process only requires a chunk of each vector X and Y and uses his process rank to offset inside the RMA windows the portion of vector to copy and targeted to partial calculations. Once each process calculates the partial dotprod value, a third RMA window is available to reduce the results into a global value stored in process with rank 0. RMA function MPI\_Accumulate over the windows dotwin is used to hold the global value.

For the current experiment 16 mpi process were involved in the benchmark of dotprod.c with two vectors of 8192 elements each.

As shown in the following table mpi rma version parallelization is slower than the pure MPI implementation.

|  |  |
| --- | --- |
| Implementation | Execution Time (s) |
| MPI | 0.000940 |
| MPI RMA | 0.004435 |

See code dotprod \_mpi\_rma.c and dotprod\_mpi.c for implementation details.

3. Parallelization of mxvnm.c using RMA operations

Parallelization of mxvnm using RMA operations replaced all communications operations by calling RMA functions. Three RMA windows were created xwin, ywin, matrix\_awin. RMA window xwin hold the vector x initialized by process with rank 0. There was no need to call any scattered MPI routine to provide a private copy of vector x to each process. Instead each process use the window xwin and read the copy of vector x using the method MPI\_Get. The same apply for the extraction of the submatrix belonging to each process from matrix a. Root process with rank 0 write to his internal buffer a and it’s exposed with the remote window matrix\_awin to all process involved in the calculation. Each process is able to extract the submatrix from the matix\_awin RMA window using the function MPI\_Get using their process rank and size of the submatrix. The call to MPI\_GET was wrapped by a MPI\_FWin\_fence on window matrix\_awin. What follows is that each process perform the calculation using the vector x and their local submatrix. At the end all process write their partial results that belong to more than one vector element of result vector ywin. Process stores their partial results using RMA window ywin and calling the function MPI\_Accumulate with the operation MPI\_REPLACE.

For the current experiment 16 mpi process were involved in the benchmark of multiplication of a matrix with 8192 rows and 8192 cols with a vectors of 8192 elements.

As shown in the following table mpi rma version parallelization is slower than the pure MPI implementation.

|  |  |
| --- | --- |
| Implementation | Execution Time (s) |
| MPI | 0.103398 |
| MPI RMA | 0.233135 |

4 . Parallelization of stencil.c using RMA operations

The parallelization of stencil.c using RMA operations present some implementation challenges that requires to adapt the mpi code to RMA programming model. Must of the creation of cartesian topology, energy source detection inside the tiles, creation of MPI\_Datatype and tile definition is similar to the MPI implementation but it differs notably on the buffers creation, halo exchange and the final reduction to calculate the global heat across all process tiles. First of all aold and anew buffers are associated to the same memory buffer and exposed to other process using an RMA Window object accessible to the other process that participate in the halo exchange. There is no other way of implement the halo exchange but turn aold and anew as pointers to the same buffers but in different positions. Anew point to the initial buffer position and aold point in the offset (bx+2)\*(by+2). The private buffer is associated with an RMA Window used only for halo exchange. Process write to each other RMA Window object using the method MPI\_Put for each of the vertical and horizontal halos and the start and end of the halo exchange is guarded by a MPI\_Win\_fence on the same RMA Window object grid\_win. At the end of each iteration there is an exchange on the aold and anew pointers. They basically change positions with respect to the offset (bx+2)\*(by+2) that’s the reason halo data could be placed on different places inside the internal memory buffet. The offset of the data referenced by aold is calculated as int offset = size\*((iter+1)%2); on each iteration. Process access each other window to write their halo information thanks to the exposed RMA Window object gridwin. At the end of the iterations the global heat is reduced from all process partial heat calculation using the MPI function MPI\_Accumulate on RMA window heat\_win in the target process with rank 0.

Following some details of the pure MPI stencil.c implementation.

The parallelization of code stencil.c using MPI mainly consists on split the whole grid into smaller tiles and assign each of them to an MPI process for energy calculation. The energy is calculated on each point taking into account the value of their neighbours and updated on each iteration. Initially process with rank 0 get the parameters grid\_size, energy and number of iterations. The input parameters are packed in an array of 3 elements and broadcasted to all process using MPI\_Bcast MPI function. Each process calls the topology functions to create a cartesian topology (MPI\_Cart\_create) and extract their coordinates in the global grid and find their neighbours rank (MPI\_Cart\_shift) equivalents to north, south, west an east. For each process is then calculated the size of the tile (bx, by) that belong to and check if inside the assigned tile where are the potential energy sources that can be mapped from the global\_grid. Two different MPI\_Datatype are created : north\_south\_type that represent the horizontal halo located on top of the tile and east\_west\_type the vertical halo bounding the left and right extremes of the tile. This is required due to calculate the energy on each process tile it’s important to know information about the neighbors. Then two buffers are created aold, anew. Those buffers have the size of (bx + 2)\*(by + 2) due to the required space to be filled by halo information. For each iteration the cells with energy values are updated and then there is a hallo exchange with all neighbours process. Initially top and bottom most cells are sent to the north and south neighbors and left and write most cells are sent to the west and east neighbors using the cartesian topology communicator created previously. The halo sent from each process to each neighbors is performed using non-blocking MPI calls MPI\_Isend. After the cells are sent each process update their neighbour halo information receiving the data from their neighbors using MPI\_Irecv. It’s important to highlight that the use of cartesian topology allows to easily work out the corner cases, hence a process with a missing neighbour is treated as an MPI\_PROC\_NULL. Any information from or sent to a MPI\_PROC\_NULL rank is discarded. The halo exchange happens with a different set of buffers than aold and anew. After the halo exchange have finished the halo information is updated in the aold buffer. The process proceed to calculate the heat on each of the point internal to his assigned tile. The energy is updated and the ouput belong the input for the next iteration till all the iteration complete. When the iteration finished the process with rank 0 perform a global reduce to sum the partial heat distributed across all the process tile using MPI\_Reduce and store the value in process with rank 0.

For the current experiment 16 mpi process were involved in the benchmark. The running parameters parameters for the test:

Grid Size : 16

Energy to be Injected Per Iteration: 1

Number of Iterations: 200

As shown in the following table mpi rma version of stencil parallelization is slower than the pure MPI implementation.

|  |  |
| --- | --- |
| Implementation | Execution Time (s) |
| MPI | 0.006467 |
| MPI RMA | 0.33567 |

See code stencil\_mpi.c and setencil\_mpi\_rma.c for implementation details.