LAB2 REPORT

February 4, 2020

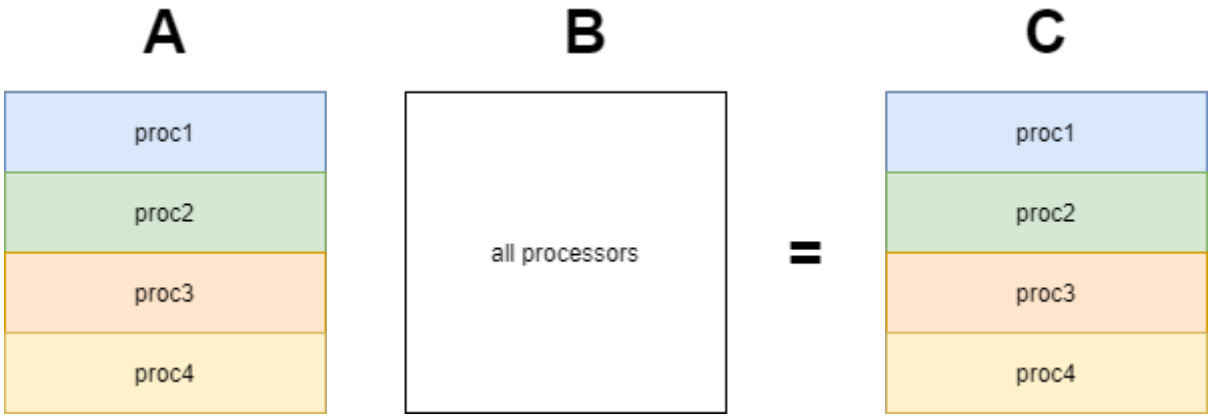
Cheng Chen (605428367)

0.1 Abstract

This report summarizes the results of parallel matrix multiplication using MPI. The configuration is explained, and different APIs and problem sizes are compared in terms of code performance.

0.2 Data and computation partition

The data and computation are partitioned as the following graph:



Matrix A is equally divided into 4 chunks, suppose the size of A is kI*kK, then every chunk is of size kI*kK/num_processor. These 4 chunks are equally distributed across 4 processors, and each processor will also have a whole copy of matrix B. Then each processor will multiply its own chunk of A with B and get a chunk of C. Finally we need to gather all the chunks of C to processor 1 so that the result can be evaluated at processor 1.

0.3 Impact of different communication APIs

I first tried blocking APIs, that is, *MPI_Send* and *MPI_Recv*. Handshake is necessary for a blocking non-buffered send/receive operation. In cases where sender and receiver do not reach communication point at similar times, there can be considerable idling overheads. The code snippets are provided as follows:

```
/**************SEND BLOCKS OF DATA*******************/

if (rank == 0){
  for (int i=1; i<numproc; i++){
    MPI_Send(&a[offset][0], aCount, MPI_FLOAT, i, 1,
                    MPI_COMM_WORLD);
    MPI_Send(b, bCount, MPI_FLOAT, i, 2, MPI_COMM_WORLD);
    offset += rows;
  }
}else{
  MPI_Recv(a_buffer, aCount, MPI_FLOAT, 0, 1, MPI_COMM_WORLD, &status);
  MPI_Recv(b_buffer, bCount, MPI_FLOAT, 0, 2, MPI_COMM_WORLD, &status);
}
```

```
if (rank != 0){
  MPI_Send(c_buffer, cCount, MPI_FLOAT, 0, 1,
                    MPI_COMM_WORLD);
}else{
  offset = rows;
  for (int i=1; i<numproc; i++){
    MPI_Recv(&c[offset][0], cCount, MPI_FLOAT, i, 1,
                    MPI_COMM_WORLD, &status);
    offset += rows;
  }
}
```

This is experimented on 4096^3 dataset and the configuration explained before. The performance is **93.8477 GFlops**.

There is a buffered blocking mechanism, that is, *MPI_Bsend*. In buffered blocking sends, the sender simply copies the data into the designated buffer and returns after the copy operation has been completed. The data is copied at a buffer at the receiving end as well. Buffering alleviates idling at the expense of copying overheads. This is not implemented as it's better to just implement the non-blocking one to compare with the blocking one.

Then I implemented the non-blocking APIs, that is, *MPI_Isend*, *MPI_Irecv*. When the message amount is large, this one incurs the least amount of overhead, as it's capable of overlapping communication overheads with useful computations. The code snippets are provided as follows:

```
if (rank != 0){
  MPI_Isend(c_buffer, cCount, MPI_FLOAT, 0, 1,
                    MPI_COMM_WORLD, &request);
}else{
  offset = rows;
  for (int i=1; i<numproc; i++){
    MPI_Irecv(&c[offset][0], cCount, MPI_FLOAT, i, 1,
                    MPI_COMM_WORLD, &request);
      offset += rows;
      MPI_Wait(&request, &status);
  }
}
```

```
MPI_Request request;
  if (rank == 0){
    for (int i=1; i<numproc; i++){
      MPI_Isend(&a[offset][0], aCount, MPI_FLOAT, i, 1,
                    MPI_COMM_WORLD, &request);
      offset += rows;
    }
  }else{
    MPI_Irecv(a_buffer, aCount, MPI_FLOAT, 0, 1, MPI_COMM_WORLD, &request);
    MPI_Wait(&request, &status);
  }

  if (rank == 0){
    for (int i=1; i<numproc; i++){
      MPI_Isend(b, bCount, MPI_FLOAT, i, 2, MPI_COMM_WORLD, &request);
      offset += rows;
    }
  }else{
    MPI_Irecv(b_buffer, bCount, MPI_FLOAT, 0, 2, MPI_COMM_WORLD, &request);
    MPI_Wait(&request, &status);
  }
```

This is experimented on 4096^3 dataset and the configuration explained before. The performance is **46.4898 GFlops**. Note that it's worse than blocking one. The reason is that, as there are only 4 processors and the information exchange is not large, the non-blocking mechanism actually involves more information tracking, checking and waiting steps, which incurs more overhead.

As a conclusion, the simple blocking mechanism will be used in this lab.

0.4 Performance Summary

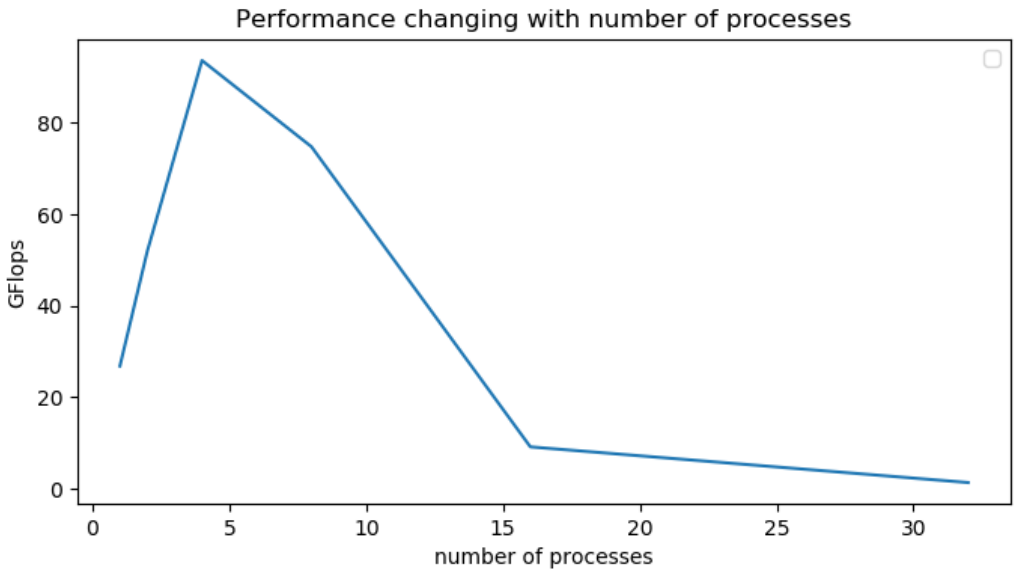
The performance is measured in terms of GFlops:

1024^3	2048^3	4096^3
42.459	76.771	93.8477

There is a huge difference in performance between different problem sizes. This is because when problem size is small, a very large proportion of the whole data can fit into cache very well, so there will be less benefit from blocking while the complexity of blocking becomes outstanding; on the other hand, when problem size is large, blocking becomes really necessary to utilize the spatial locality, so the performance is improved to a greater extent.

0.5 Code scalability

The results are shown below as a plot:



It can be shown that my code is able to produce the right result whatever the number of processors is. The performance is maximized when np=4, and it's reasonable as m5.2xlarge has 4 processors, so the performance is maximized when number of processes is set to 4. Note that the performance drops significantly when np is too large. When np is too large, some of different processes will run on the same CPU, and they have to wait for each other, and context switches will also incur a lot of latencies. As was mentioned

in the lecture, this kind of rowwise block striped decomposition has scalability function C^2p, which is linear to the number of processes. So it doesn't scale well as the number of processes gets large, which is also indicated by the plot.

0.6 Comparison with OpenMP implementation

Firstly, this MPI implementation involves more programming effort. In OpenMP, all the initialization of threads, task distribution, communication and synchronization are done implicitly by the different pragmas, we don't have to deal with them as programmers. But MPI requires us to write these out as code explicitly.

Secondly, the performance is much worse than OpenMP. One of the reasons is that all the processes mentioned above are implicitly optimized by OpenMP, which provides better performance. The other reason is that, OpenMP only creates multiple threads within a same process, which are light-weight and incurs less overhead. They share the same memory space, so no message transferring between different processes is required.