IN3200/IN4200: Dissecting Home Exam 2

Spring 2021

Objectives of home exam 2

- Exposure to a computational step in real-world machine learning
- A simple example of parallelism identification
- A real case of collaboration and data exchange on a distributed-memory system
- Hands-on experience of MPI programming

Starting point: serial single-layer convolution

```
void single_layer_convolution (int M, int N, float **input,
                               int K, float **kernel,
                               float **output)
{
  int i,j,ii,jj;
  double temp;
  for (i=0; i<=M-K; i++)
    for (j=0; j<=N-K; j++) {
       temp = 0.0;
       for (ii=0; ii<K; ii++)
         for (jj=0; jj<K; jj++)
           temp += input[i+ii][j+jj]*kernel[ii][jj];
       output[i][j] = temp;
    }
}
```

input: $M \times N$ array, output: $(M - K + 1) \times (N - K + 1)$ array

Parallelism & work partitioning

The values of the output array can be computed **independently**, so there is ample parallelism.

For simplicity, we choose a 1D block-wise decomposition of the total computational work. Specifically, each MPI process is responsible for computing a contiguous block of rows in the output array.

The in total $N - K + 1$ rows of the output array are evenly assigned to the processess. On each process, we have

```
my_output_num_rows = (M-K+1)*(my_rank+1)/num_procs
                   - (M-K+1)*my_rank/num_procs;
```

Data structure per MPI process

When the value of `my_output_num_rows` is decided per MPI process, two local 2D arrays are allocated as follows:

```
float **my_input, **my_ouput;
allocate_array2D (&my_input, my_output_num_rows+K-1, N);
allocate_array2D (&my_output, my_output_num_rows, N-K+1);
```

Note: The 2D array `my_input` has $K-1$ more rows than `my_output`, also $K-1$ more columns.

Distributing the global 2D "input" array

Each process now needs to get a corresponding part of the global 2D "input" array, which is only available on process 0. The `MPI_Scatterv` function is best suited for this purpose.

```
int *counts=NULL, *displs=NULL;

if (my_rank==0) {
  counts = (int*)malloc(num_procs*sizeof(int));
  displs = (int*)malloc(num_procs*sizeof(int));
  for (i=0; i<num_procs; i++) {
    displs[i] = ((M-K+1)*i/num_procs)*N;
    counts[i] = ((M-K+1)*(i+1)/num_procs + K-1)*N - displs[i];
  }
}

float *sendbuf = (my_rank==0) ? input[0] : NULL;
int recvcount = (my_output_num_rows+K-1)*N;
/* Must be called by all MPI processes */
MPI_Scatterv (sendbuf, counts, displs, MPI_FLOAT,
              my_input[0], recvcount, MPI_FLOAT, 0, MPI_COMM_WORLD);
```

Some important comments

- We have assumed that all the 2D arrays have an underlying **contiguous** 1D memory storage. For example, `my_input[0]` works perfectly as the "receiver buffer".
- Note that the global 2D "input" array is **empty** on all processes except on rank 0. Using `input[0]` on processes with rank>0 will thus cause segmentation fault!
- The help arrays, `displs` and `counts`, are only significant on process 0. (They can remain empty on all other processes.)
- The values received by the processes have **overlap** between them (as desired). Strictly speaking, this may not be fully compatible with the MPI standard of `MPI_Scatterv`, but it works fine in reality.

To strictly follow MPI standard of MPI_Scatterv ...

"To be on the safe side", we can adopt the following code:

```
if (my_rank==0) {
  ...
  for (i=0; i<num_procs; i++) {
    displs[i] = ((M-K+1)*i/num_procs)*N;
    counts[i] = ((M-K+1)*(i+1)/num_procs)*N - displs[i];
  }
  counts[num_procs-1] += (K-1)*N; // more to receive on last process
}

float *sendbuf = (my_rank==0) ? input[0] : NULL;
int recvcount = my_output_num_rows*N;
if (my_rank==num_procs-1)
  recvcount += (K-1)*N;
/* Distribution without overlap in the received values */
MPI_Scatterv (sendbuf, counts, displs, MPI_FLOAT,
              my_input[0], recvcount, MPI_FLOAT, 0, MPI_COMM_WORLD);

/* Additional upward "linear shift" communication */
int dest = (my_rank==0) ? MPI_PROC_NULL : (my_rank-1);
int source = (my_rank==(num_procs-1)) ? MPI_PROC_NULL : (my_rank+1);
MPI_Status status;
MPI_Sendrecv (my_input[0], (K-1)*N, MPI_FLOAT, dest, 101,
              my_input[my_output_num_rows], (K-1)*N, MPI_FLOAT, source, 101,
              MPI_COMM_WORLD, &status);
```

"Saving memory usage" on process 0

Actually, process 0 can directly use the global input array, and directly compute inside the global output array!

```
float **my_input, **my_ouput;
if (my_rank==0) {
  my_input = input;
  my_output = output
}
else {
  allocate_array2D (&my_input, my_output_num_rows+K-1, N);
  allocate_array2D (&my_output, my_output_num_rows, N-K+1);
}
```

Process 0 doesn't need to scatter data to itself

- Make sure that `counts[0]=0` on process 0 before every process calls `MPI_Scatterv`.
- Make sure that process 0 does not participate in the additional upward "linear shift" communication.

Computation now takes place on each process

When the global 2D "input" array is properly distributed to all the other processes, serial computation on each process can take place:

```
single_layer_convolution (my_output_num_rows+K-1, N, my_input,
                          K, kernel, my_output);
```

"Stitching the computed pieces together"

```
if (my_rank==0) {
  displs[0] = counts[0] = 0;
  for (i=1; i<num_procs; i++) {
    displs[i] = ((M-K+1)*i/num_procs)*(N-K+1);
    counts[i] = ((M-K+1)*(i+1)/num_procs)*(N-K+1) - displs[i];
  }
}

float *recvbuf = (my_rank==0) ? output[0] : NULL;
int sendcount = (my_rank==0) ? 0 : (my_output_num_rows*(N-K+1));
/* Must be called on every process */
MPI_Gatherv (my_output[0], sendcount, MPI_FLOAT,
             recvbuf, counts, displs, MPI_FLOAT,
             0, MPI_COMM_WORLD);
```

Double-layer convolution

The result of a double-layer convolution should be **as if** two single-layer convolutions are done in succession:

```
single_layer_convolution (M, N, input,
                          K1, kernel1, intermediate);
single_layer_convolution (M-K1+1, N-K1+1, intermediate,
                          K2, kernel2, output);
```

The 2D "intermediate" array has dimension $(M - K_1 + 1) \times (N - K_1 + 1)$

Parallel implementation of double-layer convolution

Steps:

- Distributing the global "input" array among the processes, with $K_1 - 1$ rows of overlap;
- Each process does a serial single-layer convolution to compute `my_intermediate` as its contribution to the global "intermediate" array (which only exists logically);
- Using "linear shift" to communicate $K_2 - 1$ rows of `my_intermediate` to the upward neighbor;
- Each process does another serial single-layer convolution to compute its contribution to the global "output" array;
- Using `MPI_Gatherv` to stitch all the pieces together.

Some points to ponder

- The global 2D "intermediate" array does not need to be stitched together, and then distributed! This will be a waste of communication and memory storage.
- There is no need to physically allocate the local array `my_output`, which can reuse the storage of the local array `my_input`.
- It is possible to avoid the "linear shift" communication of the "intermediate" result, by letting each processor compute $K_2 - 1$ rows extra — trading communication with excessive computation.
- It is even possible to avoid the intermediate array altogether, by merging the two separate kernels into a single $(K_1 + K_2 - 1) \times (K_1 + K_2 - 1)$ kernel. Then, a single-layer convolution is sufficient. (This strategy is not necessarily more efficient!)