MPI 9

CSCI 4850/5850 High-Performance Computing

Spring 2018

Tae-Hyuk (Ted) Ahn

Department of Computer Science
Program of Bioinformatics and Computational Biology
Saint Louis University
Learning Objectives

- Learn about how to submit multi jobs on cluster
- Learn about parallel matrix-matrix multiplication and advanced Cannon’s algorithm
Running multiple jobs

- [http://www.nersc.gov/users/computational-systems/cori/running-jobs/example-batch-scripts/](http://www.nersc.gov/users/computational-systems/cori/running-jobs/example-batch-scripts/)
Running multiple jobs example

#!/bin/bash

#SBATCH --export=ALL
#SBATCH -n 8
#SBATCH --partition=defq,med_mem,hi_mem

for i in sim/*; do
    mkdir $i/$SLURM_JOB_NAME
    integ.py ecoli.fa viral/`basename $i` > $i/$SLURM_JOB_NAME/g.fa

    wgsim -N 1000000 -1 150 -2 150 $i/$SLURM_JOB_NAME/g.fa $i/$SLURM_JOB_NAME/l.fq $i/$SLURM_JOB_NAME/r.fq

    bwa mem -a -T 0 -t 8 index/all_viral.fa $i/$SLURM_JOB_NAME/l.fq $i/$SLURM_JOB_NAME/r.fq |\samtools view -b | samtools sort -@8 - > $i/$SLURM_JOB_NAME/aln.bam

    samtools index $i/$SLURM_JOB_NAME/aln.bam

    ./m1.py $i/$SLURM_JOB_NAME/aln.bam > $i/$SLURM_JOB_NAME/stats

    rm -rf $i/$SLURM_JOB_NAME/*.*
done
Matrix-Matrix multiplication

Multiplying two square matrices:
\[ C \leftarrow A \times B, \quad A, B, C \in \mathbb{R}^{n \times n} \]
can be achieved by calculating each element of the result matrix as

\[ c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}, \]

where \( a_{ij}, b_{ij} \) and \( c_{ij} \) are the element on \( i^{th} \) row and \( j^{th} \) column of \( A, B \) and \( C \) respectively.

Pseudocode:

```
1: Procedure MAT-MULT (A, B, C)
2: begin
3:   for i ← 0 to n − 1 do
4:       for j ← 0 to n − 1 do
5:           begin
6:             c_{ij} = 0
7:             for k ← 0 to n − 1 do
8:                 c_{ij} += a_{ik} \cdot b_{kj}
9:           endfor
10:   end
11: end MAT-MULT
```

Since we assume that floating point multiplications take one time unit, and that floating point additions are very fast (they take zero time units), the time needed for the serial algorithm to complete is expressed as \( \Theta(n^3) \).
Parallel Matrix Multiplication

Write the parallel matrix multiplication using 1-D partitioning.

- Consider a homogeneous parallel machine with the number of processors \( p \) a perfect (i.e., \( \sqrt{p} \) is an integer). \( P_{i,j} \) is the processor row \( i \) and column \( j \) where \( 0 \leq i,j \leq \sqrt{p} - 1 \).
- We consider only the simple case where the matrix dimension \( n \) is a multiple of \( \sqrt{p} \) (i.e., \( n/\sqrt{p} \) is an integer).
- Implement the program using collective communications such as MPI_Scatter, MPI_Bcast, and MPI_Gather. For example, scatter A matrix and broadcast B matrix to the processes. Compute partial multiplication and gather partial C into C.
Cannon’s Algorithm

1: Set $c_{ij} = 0$, for $\forall i$ and $j$
2: Skew $A$: for $i = 0 : (\sqrt{p} - 1)$
3: left circular shift $i^{th}$ sub-block row of $A$ by $i$
4: Skew $B$: for $j = 0 : (\sqrt{p} - 1)$
5: up circular shift $j^{th}$ sub-block column of $B$ by $j$
6: for $i = 0 : (\sqrt{p} - 1)$
7: Each processor multiplies the current sub-block of $A$ by the current sub-block of $B$
8: and adds the result to the sub-block of $C$ of the processor
9: Roll $A$: left circular shift sub-blocks of $A$ by 1
10: Roll $B$: up circular shift sub-blocks of $B$ by 1
11: end

Example:

<table>
<thead>
<tr>
<th>Initializing</th>
<th>Skewing</th>
<th>Shifting</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A(0,0)$ $A(1,0)$ $A(2,0)$</td>
<td>$A(0,0)$ $A(1,0)$ $A(2,0)$</td>
<td>$A(0,1)$ $A(1,2)$ $A(2,1)$</td>
</tr>
<tr>
<td>$A(0,1)$ $A(1,1)$ $A(2,1)$</td>
<td>$A(1,1)$ $A(2,1)$ $A(0,1)$</td>
<td>$A(1,0)$ $A(2,0)$ $A(0,0)$</td>
</tr>
<tr>
<td>$A(0,2)$ $A(1,2)$ $A(2,2)$</td>
<td>$A(2,2)$ $A(0,2)$ $A(1,2)$</td>
<td>$A(2,1)$ $A(0,1)$ $A(1,0)$</td>
</tr>
</tbody>
</table>

Initial $A$, $B$

A, B initial alignment

A, B after shift step 1

A, B after shift step 2
Cannon’s Algorithm

```
// initialize matrices
if (rank == root && isValid == true) {

    // set up cartesian topology
    int ndims, dims[2], sqrt_np;
    bool periods[2], reorder;
    ndims = 2; // 2D matrix/grid
    dims[0] = dims[1] = sqrt_np = sqrt(np); // dimensions
    periods[0] = periods[1] = true; // wraparound both column and row
    reorder = true; // allows processes reordered for efficiency

    // MPI new Cartcomm (grid_comm) by Create_cart
    MPI::Cartcomm grid_comm = MPI::COMM_WORLD.Create_cart(ndims, dims, periods, reorder);

    // get the rank and coordinates with respect to the new topology
    int grid_rank, grid_coords[2];
    grid_rank = grid_comm.Get_rank();
    grid_comm.Get_coords(grid_rank, ndims, grid_coords);

    // compute ranks of the up and left shifts
    int lefrank, rightrank, uprank, downrank;
    //grid_comm.Shift(0, 1, uprank, downrank);
    //grid_comm.Shift(1, 1, lefrank, rightrank);
    //cout << "Process: " << rank << " > lefrank: " << lefrank << ", rightrank: " << rightrank << endl;
    //cout << "Process: " << rank << " > uprank: " << uprank << ", downrank: " << downrank << endl;
    grid_comm.Shift(0, -1, downrank, uprank);
    grid_comm.Shift(1, -1, rightrank, lefrank);

    // determine the dimension of the local matrix block
    int block_n, block_count;
    block_n = n / sqrt_np;
    block_count = block_n * block_n;
```
```c
// determine the dimension of the local matrix block
int block_n, block_count;
block_n = n / sqrt_np;
block_count = block_n * block_n;

// define MPI block datatype and vector
MPI::Datatype block = MPI::FLOAT.Create_vector(block_n, block_n, n).Create_resized(0,
sizeof(float));
block.Commit(); // commit the defined datatype

// prepare sendcounts and displs for MPI Scatterv
int sendcounts[np], displs[np];
for (int i = 0; i < sqrt_np; i++) {
    for (int j = 0; j < sqrt_np; j++) {
        displs[i*sqrt_np + j] = (i * sqrt_np * block_n + j) * block_n;
        sendcounts[i*sqrt_np + j] = 1;
    }
}

// send block using scatterv
float block_A[block_n][block_n], block_B[block_n][block_n], block_C[block_n][block_n];
for (int i = 0; i < block_n; i++) {
    for (int j = 0; j < block_n; j++) {
        block_A[i][j] = block_B[i][j] = block_C[i][j] = 0;
    }
}
//MPI::COMM_WORLD.Scatterv(A, sendcounts, displs, block, block_A, block_count, MPI::FLOAT, 0);
grid_comm.Scatterv(A, sendcounts, displs, block, block_A, block_count, MPI::FLOAT, 0);
```
Cannon’s Algorithm

// Cannon Step1: skewing
int src, dst;

// shift all blocks in A to the left by i steps
grid_comm.Shift(1, -grid.coords[0], src, dst);
grid_comm.Sendrecv_replace(block_A, block_count, MPI::FLOAT, dst, 1, src, 1);

// shift all blocks in B up by j steps

// Cannon Step2: calculate and shifting
for (int i = 0; i < sqrt_np; i++) {
    for (int row = 0; row < block_n; row++) {
        for (int col = 0; col < block_n; col++) {
            for (int k = 0; k < block_n; k++) {
                block_C[row][col] += block_A[row][k] * block_B[k][col];
            }
        }
    }
}
grid_comm.Shift(1, -1, src, dst); // left 1
grid_comm.Sendrecv_replace(block_A, block_count, MPI::FLOAT, dst, 0, src, 0);
grid_comm.Shift(0, -1, src, dst); // up 1
grid_comm.Sendrecv_replace(block_B, block_count, MPI::FLOAT, dst, 0, src, 0);

// gather the partial results