Parallel and Distributed Algorithms and Programs
TP n°4 - Parallel SUMMA Matrix-Matrix Multiplication

Julien Braine Laureline Pinault

6 november 2019

Scalable Universal Matrix Multiplication Algorithm (SUMMA) is one of the most popular and intuitive methods for parallel matrix-matrix multiplication. For simplicity, in this exercise, we will only be multiplying $N \times N$ matrices A and B to obtain another $N \times N$ matrix C. We will use $P = p \times p$ processors, and assume again for simplicity that p divides N. In SUMMA, the matrices A, B, and C are split into $p \times p$ submatrices. For instance, for $p = 2$, A is split into submatrices A_{11}, A_{12}, A_{21}, and A_{22} of size $N/2 \times N/2$ each. In general, each process with index (i, j) owns the corresponding submatrices A_{ij}, B_{ij}, and C_{ij}.

Algorithm 1 SUMMA matrix-matrix multiplication

Input: Matrices A, B, C of size $N \times N$
$P = p \times p$ processors
Output: $C = AB$ is computed.
1: Distribute matrices so that the process p_{ij} owns the matrices A_{ij}, B_{ij}, and C_{ij}.
2: **for** $k = 1 \dots p$ **do**
3: For all $i = 1 \dots p$, broadcast the matrix A_{ik} as A_{temp} to the process row $p_{i1} \dots p_{ip}$.
4: For all $i = 1 \dots p$, broadcast the matrix B_{ki} as B_{temp} to the process column $p_{1i} \dots p_{pi}$.
5: At each process p_{ij}, perform the local matrix multiplication update $C_{ij} = C_{ij} + A_{temp}B_{temp}$.

Note that in Algorithm 1, each matrix block of A is broadcasted to the corresponding processor row, whereas each matrix block of B is broadcasted to the corresponding processor column. In order to perform these row-wise and column-wise broadcasts, we need to create MPI communicators for each row and column of the processor grid. Generating new communicators using `MPI_Comm_split` is very simple:

```
int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)
```

- `comm` is the communicator we want to split.
- `color` is an integer which determines in which subset of process we are. New communicator will include processes sharing the same color.
- `key` is an integer used to determine the rank of a process in the new communicator. Two processes with the same color will end up in the same communicator and there relative rank will be determined by this key.
- `newcomm` is a pointer to the new communicator being created.

Note that a communicator is still available after you split it. You may split `MPI_COMM_WORLD` however you want, it will still be available for any operation (including splitting it differently to create yet another communicator).

Part 1

Parallel SUMMA using splitted communicators

Question 1

a) As discussed, implementing Algorithm 1 requires forming a communicator for each row and column of the processor grid. We can do this by splitting the default communicator `MPI_COMM_WORLD` properly. How can this be done? What `color` and `key` values should we use? Figure this out and form the row and the column communicators.

b) Instead of creating $N \times N$ matrices and distributing them, for simplicity, we will create the local submatrices of each process using the provided function

```
createMatrix(double **pmat, int nrows, int ncols, char *init).
```

Here, we provide a pointer to a double pointer (which will point to the created matrix), the number of rows and the columns of the matrix to be created, and the method of initialization of matrix elements. Providing the string "random" as `init` will initialize each matrix element randomly, whereas giving "zero" will initialize each element to 0. At each process, create the local matrices `Aloc`, `Bloc` and `Cloc` of size $(N/p) \times (N/p)$. Make sure to initialize `Aloc` and `Bloc` randomly, and `Cloc` with zeros.

c) We provide the function

```
multiplyMatrix(double *a, double *b, double *c, int m, int k, int n)
```

to perform the multiplication $C \leftarrow C + AB$ where the matrices A, B, and C are pointed by `a`, `b`, and `c`, and the matrices are of size $m \times k$, $k \times n$, and $m \times n$, respectively. Using this function, and the row/column communicators that you created, implement the SUMMA algorithm provided in Algorithm 1.

d) Measure the performance of your implementation using SMPI for $N = 1024$ and P up to 64 (using a 8×8 processor grid). You can create a grid topology of $n \times n$ processors by typing

```
python smpi-create-grid.py n 100 100 100Gbps 1us
```

with the other parameters as usual (speed, bandwidth, latency). How well does your algorithm scale? Try to change the network bandwidth, and see when it starts to lose scalability.

Make sure to backup all your implementations as they might be useful later on!