High Performance Linear Algebra

Lecture 12: ScaLAPACK and Distributed BLAS level 3

Ph.D. program in High Performance Scientific Computing

Fabio Durastante **Pasqua D'Ambra** **Salvatore Filippone**

Thursday 29, 2026 — 16.00:18.00

Dipartimento
di Matematica
Università di Pisa

Last time on High Performance Linear Algebra

We have

- We have implemented the basic routines for distributed matrices and vectors,
- We have implemented Level-1 BLAS operations for distributed matrices and vectors,
- We have implemented the GEMV operation for distributed matrices and vectors.

The plan for today is to:

- Introduce ScaLAPACK,
- Implement the Level-3 BLAS operation GEMM for distributed matrices.

Table of Contents

2 PBLAS and ScaLAPACK

The PBLAS and ScaLAPACK libraries

What we have done so far is to implement some of the routines of the PBLAS library, which is the distributed memory version of the BLAS library.

- ScaLAPACK is designed to mirror LAPACK, relying on a **P**arallel **BLAS** (PBLAS) interface that stays close to BLAS.
- Only one substantially new PBLAS routine is added: distributed matrix transposition.

Goal: Provide a distributed-memory standard like BLAS for shared memory.

The PBLAS and ScaLAPACK libraries

What we have done so far is to implement some of the routines of the PBLAS library, which is the distributed memory version of the BLAS library.

- ScaLAPACK is designed to mirror LAPACK, relying on a **P**arallel **BLAS** (PBLAS) interface that stays close to BLAS.
- Only one substantially new PBLAS routine is added: distributed matrix transposition.

Goal: Provide a distributed-memory standard like BLAS for shared memory.

2D block-cyclic layout

- PBLAS matrices use a 2D block-cyclic distribution.
- Distribution parameters are stored in an array descriptor—instead than in the modern object-oriented style we have used.

Distributed matrix descriptors

Descriptor fields

1. Number of rows
2. Number of columns
3. Row block size (Section 2.5)
4. Column block size (Section 2.5)
5. Process row of first row
6. Process column of first column
7. BLACS context
8. Leading dimension of the local array

BLACS contexts

A **BLACS context** defines a communication universe.

- Each distributed matrix is associated with a BLACS context.
- Different contexts allow independent communication universes.
- All descriptors in a PBLAS call must share the same context.
- This allows modularity in programs using multiple distributed matrices.

In our *modern implementation*, we can think of the BLACS context as an object storing the MPI communicator and the process grid information.

BLAS vs PBLAS: DGEMM vs PDGEMM

Comparing two routines for matrix-matrix multiplication

BLAS

```
CALL DGEMM(TRANSA, TRANSB, M, N, K,
           ALPHA, A(IA, JA), LDA,
           B(IB, JB), LDB, BETA,
           C(IC, JC), LDC)
```

PBLAS

```
CALL PDGEMM(TRANSA, TRANSB, M, N, K,
            ALPHA, A, IA, JA, DESC_A,
            B, IB, JB, DESC_B, BETA,
            C, IC, JC, DESC_C)
```

- DGEMM uses A(IA, JA) to specify the submatrix.
- PDGEMM requires IA, JA, and DESC_A to locate the global submatrix.
- The same applies to B and C with DESC_B and DESC_C.

We still need an MPI communicator!

In order to create a **BLACS context**, we first need to create an **MPI communicator**.

```
call MPI_Init(ierr)
call MPI_Comm_size(MPI_COMM_WORLD,
↪  nprocs, ierr)
call MPI_Comm_rank(MPI_COMM_WORLD,
↪  myrank, ierr)
```

The generic structure of a BLACS application is as follows:

1. Initialize MPI
2. Initialize BLACS
3. Create a process grid
4. Query process coordinates
5. Perform communication
6. Destroy grid and exit BLACS
7. Finalize MPI

We still need an MPI communicator!


```
integer :: ictxt, nprow, npcol
call blacs_get(-1, 0, ictxt)
call blacs_gridinit(ictxt, 'R',
↪  nprow, npcol)
```

Where

- `ictxt` is the BLACS context identifier,
- `blacs_get` initializes the BLACS system,
- `blacs_gridinit` creates a process grid with `nprow` rows and `npcol` columns.

The generic structure of a BLACS application is as follows:

1. Initialize MPI
2. Initialize BLACS
3. Create a process grid
4. Query process coordinates
5. Perform communication
6. Destroy grid and exit BLACS
7. Finalize MPI

We still need an MPI communicator!

We can then query the process coordinates in the grid:

```
integer :: myrow, mycol
call blacs_gridinfo(ictxt, nprow,
↪  npcol, myrow, mycol)
```

Where

- `myrow` is the row coordinate of the process,
- `mycol` is the column coordinate of the process.

The generic structure of a BLACS application is as follows:

1. Initialize MPI
2. Initialize BLACS
3. Create a process grid
4. Query process coordinates
5. Perform communication
6. Destroy grid and exit BLACS
7. Finalize MPI

Row major vs Column major

The two common ways to map a 1D array to a 2D array are:

```
! Row major mapping
index = (i-1)*ncols + (j-1) + 1
```

0	1	2	3
4	5	6	7
8	9	10	11

```
! Column major mapping
index = (j-1)*nrows + (i-1) + 1
```

0	3	6	9
1	4	7	10
2	5	8	11

Row major:

```
call blacs_gridinit(ictxt, 'R', nprow, npcol)
```

Row major vs Column major

The two common ways to map a 1D array to a 2D array are:

```
! Row major mapping
index = (i-1)*ncols + (j-1) + 1
```

0	1	2	3
4	5	6	7
8	9	10	11

```
! Column major mapping
index = (j-1)*nrows + (i-1) + 1
```

0	3	6	9
1	4	7	10
2	5	8	11

Column major:

```
call blacs_gridinit(ictxt, 'C', nprow, npcol)
```

A complete example of initialization


```
integer :: ierr, nprocs, myrank
integer :: ctxt, nrows, ncols, myrow, mycol
integer :: info
! Initialize MPI
call MPI_Init(ierr)
call MPI_Comm_size(MPI_COMM_WORLD, nprocs, ierr)
call MPI_Comm_rank(MPI_COMM_WORLD, myrank, ierr)
! Initialize BLACS
call blacs_get(-1, 0, ctxt)
! Create a process grid
nrows = int(sqrt(real(nprocs)))
ncols = nprocs / nrows
call blacs_gridinit(ctxt, 'C', nrows, ncols)
! Query process coordinates
call blacs_gridinfo(ctxt, nrows, ncols, myrow, mycol)
```

If we compile and execute this code

Compiling with

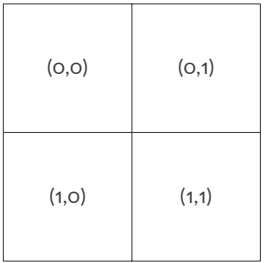
```
mpifort -o blacs_init blacs_init.f90 -lscalapack
```

And executing with

```
mpirun -np 4 ./blacs_init
```

We could obtain the following output:

```
BLACS grid: 2 x 2
Processor 3 is at (1,1)
Processor 0 is at (0,0)
Processor 1 is at (1,0)
Processor 2 is at (0,1)
```



Where to find BLACS, PBLAS, and ScaLAPACK

- BLACS, PBLAS, and ScaLAPACK are usually provided as part of high-performance linear algebra libraries such as
  - Intel MKL,
  - AMD ACML,
  - Netlib ScaLAPACK.
- The latter can also be built from source code available from the Netlib repository:
  - http://www.netlib.org/scalapack/

They can be installed via Spack as well:

```
spack install netlib-scalapack
```

or

```
spack install intel-oneapi-mkl
```

PBLAS operations

The **PBLAS library** provides distributed memory implementations of the Level-1, Level-2, and Level-3 BLAS operations.

- Level-1 PBLAS operations include vector-vector operations such as **P?AXPY**.
- Level-2 PBLAS operations include matrix-vector operations such as **P?GEMV**.
- Level-3 PBLAS operations include matrix-matrix operations such as **P?GEMM**.

We can now try using the P?GEMV operation, and try to measure its performance.

The P?GEMV operation

The **P?GEMV** operation computes the matrix-vector product

$$y \leftarrow \alpha Ax + \beta y,$$

where A is a distributed matrix, and x and y are distributed vectors.

- The operation is called via the `PDGEMV` routine.
- The routine requires the descriptors of the distributed matrix and vectors.

The routine signature is as follows:

```
CALL PDGEMV(TRANS, M, N, ALPHA, A, IA, JA, DESC_A, X, IX, 1, DESC_X,
            BETA, Y, IY, 1, DESC_Y)
```

The P?GEMV operation: parameters


```
CALL PDGEMV(TRANS, M, N, ALPHA, A, IA, JA, DESC_A, X, IX, 1, DESC_X,
            BETA, Y, IY, 1, DESC_Y)
```

- `TRANS` specifies whether to use A or A^T,
- `M` and `N` are the number of rows and columns of A,
- `ALPHA` and `BETA` are scalars,
- `A` is the local array containing the local pieces of A,
- `IA` and `JA` are the row and column indices of the first element of the submatrix of A,
- `DESC_A` is the descriptor of A,
- `X` is the local array containing the local pieces of x,
- `IX` is the index of the first element of the subvector of x,
- `DESC_X` is the descriptor of x,
- `Y` is the local array containing the local pieces of y,
- `IY` is the index of the first element of the subvector of y,
- `DESC_Y` is the descriptor of y.

Testing PDGEMV

To test the `PDGEMV` routine, we can write a simple Fortran program that:

1. Build the test program and link against ScaLAPACK.
2. Run with a square number of MPI ranks.
3. Choose m, n, and block size nb via command-line arguments.

Then we can call `PDGEMV` to perform the matrix-vector multiplication.

The code to compile and run the test program is something on the lines of:

```
mpifort -O3 -o test_pdegemv test_pdegemv.f90 -lscalapack
mpirun -np 4 ./test_pdegemv 4000 4000 128
```

Reading command-line arguments

We can read command-line arguments in Fortran as follows:

```
m = 4000
n = 4000
nb = 128
nreps = 10

arg_count = command_argument_count()
if (arg_count >= 1) then
  call get_command_argument(1, arg)
  read(arg, *) m
end if
if (arg_count >= 2) then
  call get_command_argument(2, arg)
  read(arg, *) n
end if
if (arg_count >= 3) then
  call get_command_argument(3, arg)
  read(arg, *) nb
end if
if (arg_count >= 4) then
  call get_command_argument(4, arg)
  read(arg, *) nreps
end if
```

Which allows us to set m, n, nb, and the number of repetitions $nreps$ for the matrix-vector multiplication.

Initializing distributed environment

We use the init code shown before to initialize MPI and BLACS.

```
call MPI_Init(ierr)
call MPI_Comm_rank(MPI_COMM_WORLD, world_rank, ierr)
call MPI_Comm_size(MPI_COMM_WORLD, world_size, ierr)
nprocs_per_dim = int(sqrt(dble(world_size)))
if (nprocs_per_dim * nprocs_per_dim /= world_size) then
  if (world_rank == 0) then
    print *, 'Error: number of processes must be a perfect square.'
  end if
  call MPI_Finalize(ierr)
  stop 1
end if
```

and check that the number of processes is a **perfect square**.

Initializing distributed environment

Then the BLACS initialization follows:

```
call blacs_get(-1, 0, ictxt)
nprow = nprocs_per_dim
npcol = nprocs_per_dim
call blacs_gridinit(ictxt, 'R', nprow, npcol)
call blacs_gridinfo(ictxt, nprow, npcol, myrow, mycol)

if (myrow == -1 .or. mycol == -1) then
  call blacs_exit(1)
  call MPI_Finalize(ierr)
  stop 1
end if
```

Creating distributed matrices and vectors

We can now create the distributed matrix A and vectors $\mathbf{x}$ and $\mathbf{y}$:

```
mloc = numroc(m, nb, myrow, 0, nprow)
nloc = numroc(n, nb, mycol, 0, npcol)
lldA = max(1, mloc)

xloc = numroc(n, nb, myrow, 0, nprow)
yloc = numroc(m, nb, myrow, 0, nprow)
lldX = max(1, xloc)
lldY = max(1, yloc)
```

Where we use `NUMROC` to compute the local sizes of the distributed arrays.

The NUMROC utility

The NUMROC utility computes the number of rows or columns of a distributed matrix or vector owned by a given process.

```
integer function numroc(n, nb, iproc, isrcproc, nprocs)
```

Where

- `n` is the global number of rows or columns,
- `nb` is the block size,
- `iproc` is the coordinate of the process in the grid,
- `isrcproc` is the coordinate of the process owning the first row or column,
- `nprocs` is the number of processes in the grid dimension.

Creating the descriptors

The descriptors for the distributed matrix and vectors for ScaLAPACK are variables of type **integer**, defined as arrays of size 9.

```
integer :: descA(9), descX(9), descY(9)
```

We can initialize them as follows:

```
call descinit(descA, m, n, nb, nb, 0, 0, ictxt, lldA, info)
call descinit(descX, n, 1, nb, 1, 0, 0, ictxt, lldX, info)
call descinit(descY, m, 1, nb, 1, 0, 0, ictxt, lldY, info)
```

We should always check the value of `info` after each call to `descinit`, e.g.,

```
if (info /= 0) then
  if (world_rank == 0) print *, 'descinit error: ', info
  call MPI_Abort(MPI_COMM_WORLD, info, ierr)
end if
```

Allocate and fill the local arrays

Next, we allocate and fill the local arrays:

```
allocate(A(lldA, max(1, nloc)))
allocate(X(lldX, max(1, numroc(1, 1, mycol, 0, npcol))))
allocate(Y(lldY, max(1, numroc(1, 1, mycol, 0, npcol))))
```

Where we use again the `numroc` utility to compute the local sizes of the vectors.

We **fill the matrix** A with:

```
if (mloc > 0 .and. nloc > 0) then
  do j = 1, nloc
    col_global = indxl2g(j, nb, mycol, 0, npcol)
    do i = 1, mloc
      row_global = indxl2g(i, nb, myrow, 0, nprow)
      A(i, j) = dble(row_global + col_global) / dble(m + n)
    end do
  end do
end if
```

Allocate and fill the local arrays

Next, we allocate and fill the local arrays:

```
allocate(A(lldA, max(1, nloc)))
allocate(X(lldX, max(1, numroc(1, 1, mycol, 0, npcol))))
allocate(Y(lldY, max(1, numroc(1, 1, mycol, 0, npcol))))
```

Where we use again the `numroc` utility to compute the local sizes of the vectors.
We **fill the vectors $\mathbf{x}$**, and $\mathbf{y}$ with:

```
if (xloc > 0) then
  do i = 1, xloc
    row_global = indxl2g(i, nb, myrow, 0, nprow)
    X(i, 1) = 1.0d0 + dble(row_global) / dble(n)
  end do
end if
if (yloc > 0) Y(1:yloc, 1) = 0.0d0
```

Timing and synchronization

- Use `MPI_Barrier` to synchronize before and after `PDGEMV`.
- Measure elapsed time with `MPI_Wtime`.

```
call MPI_Barrier(MPI_COMM_WORLD, ierr)
t0 = MPI_Wtime()
do i = 1, nreps
  call pdgemv('N', m, n, alpha, A, 1, 1, descA, X, 1, 1, descX, 1, beta, Y,
  ↪  1, 1, descY, 1)
end do
call MPI_Barrier(MPI_COMM_WORLD, ierr)
t1 = MPI_Wtime()
elapsed_time = (t1 - t0)
```

As usual, this the `elapsed_time` contains the total time for *nreps* repetitions of the matrix-vector multiplication on each process.

Computing the performance

2 PBLAS and ScaLAPACK

We take the **worst-case time** across all processes:

```
call MPI_Reduce(elapsed_time, max_elapsed, 1, MPI_DOUBLE_PRECISION,
↪  MPI_MAX, 0, MPI_COMM_WORLD, ierr)
```

Then, on rank 0, we can compute the measurements:

```
if (myrow == 0 .and. mycol == 0) then
  gflops = (2.0d0 * dble(m) * dble(n) * dble(nreps)) / max_elapsed / 1.0d9
  avg_time = max_elapsed / dble(nreps)
  avg_gflops = (2.0d0 * dble(m) * dble(n)) / avg_time / 1.0d9
  print *, 'PDGEMV m=', m, ' n=', n, ' nb=', nb, ' procs=', world_size, '
  ↪  reps=', nreps
  print *, 'Total time (s)=', max_elapsed, ' Total GFLOPS=', gflops
  print *, 'Avg time (s)=', avg_time, ' Avg GFLOPS=', avg_gflops
end if
```

GFLOPS calculation for PDGEMV

We can compute the number of floating-point operations for the matrix-vector multiplication as follows:

$$\text{FLOPS} = 2 \cdot m \cdot n$$

Thus, the performance in GFLOPS can be computed as:

```
gflops = (2.0d0 * dble(m) * dble(n) * dble(nreps)) / max_elapsed / 1.0d9
```

Where `nreps` is the number of repetitions of the multiplication.

Similarly, the worst-case performance per repetition is:

```
avg_gflops = (2.0d0 * dble(m) * dble(n)) / avg_time / 1.0d9
```

Finalizing the distributed environment

Finally, we need to finalize BLACS and MPI:

```
if (allocated(A)) deallocate(A)
if (allocated(X)) deallocate(X)
if (allocated(Y)) deallocate(Y)
call blacs_gridexit(ictxt)
call blacs_exit(0)
```

We don't need to call `MPI_Finalize` explicitly, as it is called inside the `blacs_gridexit`.

We are finally done, and we can compile and run our test program. As for the case in the other lectures, we plan on running it on the Amelia cluster at IAC-CNR. Hence, we use the Intel Oneapi compilers, and link against Intel MKL—which provides BLACS, PBLAS, and ScaLAPACK.

Running script

The script to run the test program on Amelia is written as follows:

```
#!/usr/bin/env bash
#SBATCH --nodes=@NODES@
#SBATCH --ntasks=@TASKS@
#SBATCH --cpus-per-task=@THREADS@
#SBATCH --partition=prod-gn
#SBATCH --time=01:00:00
#SBATCH --mem=950Gb
#SBATCH --job-name=pdegemv_weak

export OMP_NUM_THREADS=1
export MKL_NUM_THREADS=1
cd /ifs/hpc/home/fdurastante/scalapacktest/launchscripts/
mpirun -np @TASKS@ ./../build/test_pdegemv @N@ @N@ @NB@ 2>&1 >
↪  ../logfiles/pdegenmv/log_t@TASKS@.log
```

Running script explanation

The script uses SLURM directives to request resource and has a number of placeholders:

- `@NODES@`: number of nodes to use,
- `@TASKS@`: number of MPI tasks to use,
- `@THREADS@`: number of threads per task,
- `@N@`: size of the matrix and vectors,
- `@NB@`: block size.

We use an auxiliary script to replace the placeholders and submit the job to SLURM:

```
./genscript.sh launch_pdegenmv.sh
```

Where `genscript.sh` replaces the placeholders and produces the launch files.

The `genscript.sh` auxiliary script

The auxiliary script `genscript.sh` is as follows:

```
#!/bin/sh
TEMPLATE="$1"
TASKS_PER_NODE=64
THREADS_PER_TASK=1
N_LOCAL=4000
NB=128
# Perfect-square task counts up to 20 nodes * 64 tasks/node = 1280 tasks
TASKS_LIST="64 144 256 400 576 900 1024 1156"
for TASKS in $TASKS_LIST; do
    # Compute number of nodes (ceiling division)
    NODES=$(( (TASKS + TASKS_PER_NODE - 1) / TASKS_PER_NODE ))
    PROCS_PER_DIM=$(LC_NUMERIC=C echo "scale=0; sqrt($TASKS)" | bc -l)
```

The `genscript.sh` auxiliary script


```
    N=$((N_LOCAL * PROCS_PER_DIM))
    OUTFILE="outscript_t${TASKS}.sh"
    sed \
        -e "s/@NODES@/${NODES}/g" \
        -e "s/@TASKS@/${TASKS}/g" \
        -e "s/@THREADS@/${THREADS_PER_TASK}/g" \
        -e "s/@N@/${N}/g" \
        -e "s/@NB@/${NB}/g" \
        "$TEMPLATE" > "$OUTFILE"
    chmod +x "$OUTFILE"
    echo "Generated $OUTFILE (tasks=$TASKS, nodes=$NODES,
    ↪  threads=$THREADS_PER_TASK, N=$N, NB=$NB)"
done
```

The `genscript.sh` auxiliary script

The script iterates over a list of perfect-square task counts

$$n_p = 64, 144, 256, 400, 576, 900, 1024, 1156,$$

computes the number of nodes required, the problem size N, and replaces the placeholders in the template script using `sed`.

The `sed` command replaces each placeholder with the corresponding value, its general form being:

```
sed -e "s/@PLACEHOLDER@/value/g" input_template > output_script
```

Where `@PLACEHOLDER@` is replaced with `value` in the `input_template`, and the result is saved in `output_script`.

The `genscript.sh` auxiliary script

We run the `genscript.sh` script as follows:

```
chmod +x genscript.sh
./genscript.sh launch_pdegemv.sh
```

Which produces the following output:

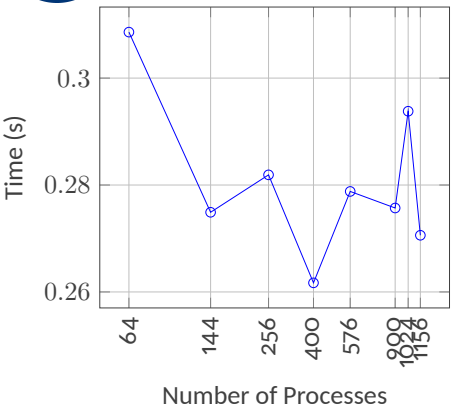
```
Generated outscript_t64.sh (tasks=64, nodes=1, threads=1, N=32000, NB=128)
Generated outscript_t144.sh (tasks=144, nodes=3, threads=1, N=48000, NB=128)
Generated outscript_t256.sh (tasks=256, nodes=4, threads=1, N=64000, NB=128)
Generated outscript_t400.sh (tasks=400, nodes=7, threads=1, N=80000, NB=128)
Generated outscript_t576.sh (tasks=576, nodes=9, threads=1, N=96000, NB=128)
Generated outscript_t900.sh (tasks=900, nodes=15, threads=1, N=120000, NB=128)
Generated outscript_t1024.sh (tasks=1024, nodes=16, threads=1, N=128000, NB=128)
Generated outscript_t1156.sh (tasks=1156, nodes=19, threads=1, N=136000, NB=128)
```

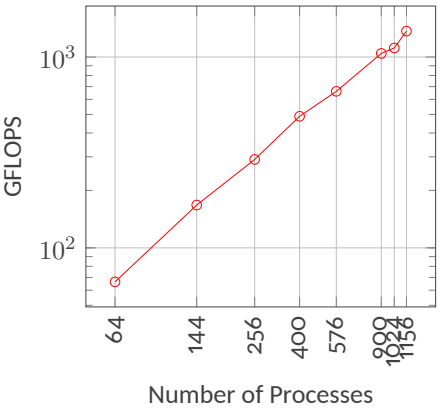
We can then submit each generated script to SLURM with:

```
sbatch outscript_t??.sh
```

Analyzing the results

2 PBLAS and ScaLAPACK





We can see that the time remains roughly constant as we increase the number of processes, while the performance increases accordingly $N =$, $NB =$.

The distributed Matrix-Matrix multiplication

- We will consider the formation of the matrix products

$$C = \alpha AB + \beta C$$
$$C = \alpha AB^T + \beta C$$
$$C = \alpha A^T B + \beta C$$
$$C = \alpha A^T B^T + \beta C$$

- These are the special cases implemented in the sequential BLAS `GEMM`.
- Assume each matrix X has dimensions $m^X \times n^X$, with $X \in \{A, B, C\}$.
- Dimensions must be compatible; we take $C \in \mathbb{R}^{m \times n}$ and the inner dimension k.

Back on the data distribution

- We consider 2D data decompositions;
- The 1D case is obtained by setting one grid dimension to 1.
- Given $X \in \{A, B, C\}^{m \times n}$ on an $r \times c$ process grid, we partition:

$$X = \begin{pmatrix} X_{00} & \cdots & X_{0(c-1)} \\ \vdots & & \vdots \\ X_{(r-1)0} & \cdots & X_{(r-1)(c-1)} \end{pmatrix}$$

- Submatrix X_{ij} is assigned to process P_{ij}.
- X_{ij} has size $m_i^X \times n_j^X$, with $\sum_i m_i^X = m$ and $\sum_j n_j^X = n$.
- Each algorithm variant enforces row/column compatibility in these dimensions: For this operation to be well-defined, we require $m^A = m$, $n^A = m^B = k$, and $n^B = n$.

Forming $C = \alpha AB + \beta C$

For simplicity, we take $\alpha = 1$ and $\beta = 0$ in our description.
If a_{ij}, b_{ij}, and c_{ij} denote the (i,j) element of the matrices, respectively, then the elements of C are given by

$$c_{ij} = \sum_{l=1}^{k} a_{il} b_{lj}.$$

Notice that:

- rows of C are computed from rows of A, and columns of C
- We hence restrict our data decomposition so that rows of A and C are assigned to the same row of nodes and columns of B and C are assigned to the same column of nodes.
- Hence, $m_i^C = m_i^A$ and $n_j^C = n_j^B$.

Basic parallel algorithm (as we have seen for the OpenMP case)

- Compute C_{ij} as a sequence of rank-one updates.
- Assign block row $\tilde{A}_i$ to process row i.
- Assign block column $\tilde{B}^j$ to process column j.

$$\tilde{A}_i = \begin{pmatrix} a_i^{(0)} & a_i^{(1)} & \cdots & a_i^{(k-1)} \end{pmatrix}, \qquad \tilde{B}^j = \begin{pmatrix} b_j^{(0)T} \\ b_j^{(1)T} \\ \vdots \\ b_j^{(k-1)T} \end{pmatrix}$$

$$C_{ij} = \sum_{t=0}^{k-1} a_i^{(t)} b_j^{(t)T}$$

Rank-one update view

Each step t:

- broadcasts $a_i^{(t)}$ along process row i.
- broadcasts $b_j^{(t)}$ along process column j.

Perform

- Local update on P_{ij}: $C_{ij} \leftarrow C_{ij} + a_i^{(t)} b_j^{(t)T}$.

$$C_{ij}^{(t+1)} = C_{ij}^{(t)} + a_i^{(t)} b_j^{(t)T}$$

1: $C_{ij} \leftarrow 0$
2: **for** $\ell = 0, \ldots, k-1$ **do**
3: broadcast $a_i^{(\ell)}$ within my row
4: broadcast $b_j^{(\ell)}$ within my column
5: $C_{ij} \leftarrow C_{ij} + a_i^{(\ell)} b_j^{(\ell)T}$
6: **end for**

Now we can analyze the cost of this algorithm, both in **terms of computation** and **communication**.

Algorithm cost: minimum spanning tree broadcast

To analyze the cost of this basic algorithm, we make some **simplifying assumptions**:

$$m_i^C = m_i^A = m/r, \qquad n_j^C = n_j^B = n/c,$$
$$n_i^A = k/c, \qquad m_j^B = k/r.$$

Since relatively little data is involved during each broadcast we assume a **minimum-spanning-tree broadcast**.

Algorithm cost: minimum spanning tree broadcast

To analyze the cost of this basic algorithm, we make some **simplifying assumptions**:

$$m_i^C = m_i^A = m/r, \qquad n_j^C = n_j^B = n/c,$$
$$n_i^A = k/c, \qquad m_j^B = k/r.$$

Since relatively little data is involved during each broadcast we assume a **minimum-spanning-tree broadcast**.

Minimum-Spanning-Tree Broadcast

A minimum-spanning-tree (MST) broadcast minimizes latency by organizing the broadcast in $\log(p)$ stages, where p is the number of processes.

- Stage 0: Process 0 sends to process 1
- Stage 1: Processes 0,1 send to processes 2,3
- Stage 2: Processes 0,1,2,3 send to processes 4,5,6,7

 And so on...

Algorithm cost: minimum spanning tree broadcast

To analyze the cost of this basic algorithm, we make some **simplifying assumptions**:

$$m_i^C = m_i^A = m/r, \qquad n_j^C = n_j^B = n/c,$$
$$n_i^A = k/c, \qquad m_j^B = k/r.$$

Since relatively little data is involved during each broadcast we assume a **minimum-spanning-tree broadcast**.

Minimum-Spanning-Tree Broadcast

Cost model for MST broadcast of data size S to p processes:

$$T_{\text{bcast}} = \lceil \log_2(p) \rceil \cdot (\alpha + S \cdot \beta)$$

Where:

- α and β are the latency and the inverse bandwidth (per unit data),
- Each stage involves one message transmission for $= \lceil \log_2(p) \rceil$ total stages.

Cost model (detailed)

The cost of the algorithm (per panel) is therefore

$$k\left[\frac{2mn}{p}\gamma + \lceil\log(c)\rceil\left(\alpha + \frac{m}{r}\beta\right) + \lceil\log(r)\rceil\left(\alpha + \frac{n}{c}\beta\right)\right].$$

The three terms inside the square brackets correspond to

- $2mn/p\,\gamma$: local rank-one update,
- $\lceil\log(c)\rceil\,(\alpha + m/r\beta)$: row broadcast of A,
- $\lceil\log(r)\rceil\,(\alpha + n/c\beta)$: column broadcast of B.

Hence the total time is

$$T(m,n,k,p) = \frac{2mnk}{p}\gamma + k\big(\lceil\log(c)\rceil + \lceil\log(r)\rceil\big)\alpha + \lceil\log(c)\rceil\frac{mk}{r}\beta + \lceil\log(r)\rceil\frac{nk}{c}\beta$$

Scalability analysis

2 PBLAS and ScaLAPACK

The cost $T(m,n,k,p)$ compares to the sequential time $2mnk\,\gamma$.

To **study scalability**, set $m = n = k$, $r = c = \sqrt{p}$ (assume p power of two).
From $T(m,n,k,p)$ the **estimated speedup** is

$$S(n,p) = \frac{2n^3\gamma}{\frac{2n^3}{p}\gamma + n\log(p)\alpha + \log(p)\frac{n^2}{\sqrt{p}}\beta} = \frac{p}{1 + \frac{p\log(p)}{2n^2}\frac{\alpha}{\gamma} + \frac{\sqrt{p}\log(p)}{2n}\frac{\beta}{\gamma}}.$$

The corresponding **parallel efficiency** is

$$E(n,p) = \frac{S(n,p)}{p} = \frac{1}{1 + \frac{p\log(p)}{2n^2}\frac{\alpha}{\gamma} + \frac{\sqrt{p}\log(p)}{2n}\frac{\beta}{\gamma}} = \frac{1}{1 + O\left(\frac{p\log p}{n^2}\right) + O\left(\frac{\sqrt{p}\log p}{n}\right)}.$$

Scalability insight

- Speedup improves with $\sqrt{p}$ process grids.
- Efficiency degrades slowly with $\log(p)$ broadcast costs.
- Increasing n with $\sqrt{p}$ keeps memory per process fixed.

$$E(n,p) = \frac{1}{1 + O\left(\frac{p\log(p)}{n^2}\right) + O\left(\frac{\sqrt{p}\log(p)}{n}\right)}$$

Scalability insight

Ignoring the $\log(p)$ term, which grows *very* slowly when p is reasonably large, we notice the following: If we increase p and we wish to maintain efficiency, we must increase n with $\sqrt{p}$. Since **memory requirements grow with** n^2, and physical memory grows linearly with p as nodes are added.

Scalability insight

- Speedup improves with $\sqrt{p}$ process grids.
- Efficiency degrades slowly with $\log(p)$ broadcast costs.
- Increasing n with $\sqrt{p}$ keeps memory per process fixed.

$$E(n,p) = \frac{1}{1 + O\left(\frac{p\log(p)}{n^2}\right) + O\left(\frac{\sqrt{p}\log(p)}{n}\right)}$$

Scalability insight

We conclude that **the method is scalable** in the following sense:
"If we maintain memory use *per node*, this algorithm will maintain efficiency, if $\log(p)$ is treated as a constant."

Moving to the pipelined algorithm

We will present the benefits of **pipelining computations and communications**.

Let us consider implementing the **broadcast** as passing of a message around the logical ring that forms the row or column.

```
subroutine RING_Bcast(data, count, type, root, comm)
 implicit none
 integer, intent(in) :: count, type, root, comm
 real(kind=real64), intent(inout) :: data(*)
 ! Local variables
 integer :: me, np, next, prev, ierr
 call MPI_Comm_rank(comm, me, ierr)
 call MPI_Comm_size(comm, np, ierr)
```

Moving to the pipelined algorithm


```
 if ( me /= root ) then
   prev = mod(me - 1 + np, np)
   call MPI_Recv(data, count, type, prev, MPI_ANY_TAG, comm,
   ↪  MPI_STATUS_IGNORE, ierr)
 end if
 if ( mod(me + 1, np) /= root ) then
   next = mod(me + 1, np)
   call MPI_Send(data, count, type, next, 0, comm, ierr)
 end if
end subroutine RING_Bcast
```

Moving to the pipelined algorithm

In this case, the time complexity becomes:

- The time required for both the first column of A_{r-1} and the first row of B^{c-1} to reach process $(r-1, c-1)$:

$$(c-1)\left(\alpha + \frac{m}{r}\beta\right) + (r-1)\left(\alpha + \frac{n}{c}\beta\right)$$

Moving to the pipelined algorithm

In this case, the time complexity becomes:

- The time required for both the first column of A_{r-1} and the first row of B^{c-1} to reach process $(r-1, c-1)$
- The time for performing the local update and passing the messages along the pipe

$$+k\left(\frac{2mn}{p}\gamma + \alpha + \frac{m}{r}\beta + \alpha + \frac{n}{c}\beta\right)$$

Moving to the pipelined algorithm

In this case, the time complexity becomes:

- The time required for both the first column of A_{r-1} and the first row of B^{c-1} to reach process $(r-1, c-1)$
- The time for performing the local update and passing the messages along the pipe
- The time for the final messages (initiated at process $(r-1, c-1)$) to reach the end of the pipe:

$$+(c-2)\left(\alpha + \frac{m}{r}\beta\right) + (r-2)\left(\alpha + \frac{n}{c}\beta\right)$$

Moving to the pipelined algorithm

In this case, the time complexity becomes:

- The time required for both the first column of A_{r-1} and the first row of B^{c-1} to reach process $(r-1, c-1)$
- The time for performing the local update and passing the messages along the pipe
- The time for the final messages (initiated at process $(r-1, c-1)$) to reach the end of the pipe
- the time for the final update at the node at the end of the pipe (process $(r-1, c-2)$ or $(r-2, c-1)$):

$$+\frac{2mn}{p}\gamma$$

Moving to the pipelined algorithm

In this case, the time complexity becomes:

- **+** The time required for both the first column of A_{r-1} and the first row of B^{c-1} to reach process $(r-1, c-1)$
- **+** The time for performing the local update and passing the messages along the pipe
- **+** The time for the final messages (initiated at process $(r-1, c-1)$) to reach the end of the pipe
- **+** the time for the final update at the node at the end of the pipe (process $(r-1, c-2)$ or $(r-2, c-1)$)
- **=** Summing all contributions, we obtain:

$$= \frac{2mn(k+1)}{p}\gamma + (k+2c-3)\left(\alpha + \frac{m}{r}\beta\right) + (k+2r-3)\left(\alpha + \frac{n}{c}\beta\right)$$

Comparing the two approaches

For the **minimum-spanning-tree broadcast** we had a cost of

$$T(m,n,k,p) = \frac{2mnk}{p}\gamma + k\big(\lceil\log(c)\rceil + \lceil\log(r)\rceil\big)\alpha + \lceil\log(c)\rceil\frac{mk}{r}\beta + \lceil\log(r)\rceil\frac{nk}{c}\beta,$$

For the **pipelined ring broadcast** we have a cost of

$$T(m,n,k,p) = \frac{2mn(k+1)}{p}\gamma + (k+2c-3)\left(\alpha + \frac{m}{r}\beta\right) + (k+2r-3)\left(\alpha + \frac{n}{c}\beta\right),$$

Notice that for large k, the "log" factors we had in the tree approach are removed.

Scalability of the pipelined approach

To **establish the scalability**, we again analyse the case where $m = n = k$ and $r = c = \sqrt{p}$.

This changes the complexity to approximately

$$\frac{2n^3}{p}\gamma + 2(n + 2\sqrt{p} - 3)\left(\alpha + \frac{n}{\sqrt{p}}\beta\right).$$

The **speedup** is

$$S(n,p) = \frac{2n^3\gamma}{\frac{2n^3}{p}\gamma + 2(n + 2\sqrt{p} - 3)\left(\alpha + \frac{n}{\sqrt{p}}\beta\right)} \approx \frac{p}{1 + \frac{p}{n^2}\alpha + \frac{\sqrt{p}}{n}\beta}$$

The corresponding **efficiency** is

$$E(n,p) = \frac{S(n,p)}{p} = \frac{1}{1 + O\left(\frac{p}{n^2}\right) + O\left(\frac{\sqrt{p}}{n}\right)}$$

Scalability of the pipelined approach

To **establish the scalability**, we again analyse the case where $m = n = k$ and $r = c = \sqrt{p}$.

The **speedup** is

$$S(n,p) = \frac{2n^3\gamma}{\frac{2n^3}{p}\gamma + 2(n + 2\sqrt{p} - 3)\left(\alpha + \frac{n}{\sqrt{p}}\beta\right)} \approx \frac{p}{1 + \frac{p}{n^2}\alpha + \frac{\sqrt{p}}{n}\beta}$$

The corresponding **efficiency** is

$$E(n,p) = \frac{S(n,p)}{p} = \frac{1}{1 + O\left(\frac{p}{n^2}\right) + O\left(\frac{\sqrt{p}}{n}\right)}$$

The $\log(p)$ term has disappeared and the method is again scalable in the sense that if we **maintain memory use per node**, this algorithm will maintain efficiency.

Blocking to further improve performance

We can reformulate the algorithm to use matrix-matrix multiplcations, instead of **rank-one updates**.

Memory acces bandwidth

Recall that matrix-matrix multiplications have a much higher **computational intensity** (flops per memory access), hence they better exploit memory hierarchies and achieve higher performance on modern architectures.

Blocking to further improve performance

We can reformulate the algorithm to use matrix-matrix multiplcations, instead of **rank-one updates**.

Memory acces bandwidth

Recall that matrix-matrix multiplications have a much higher **computational intensity** (flops per memory access), hence they better exploit memory hierarchies and achieve higher performance on modern architectures.

Blocking to further improve performance

We can reformulate the algorithm to use matrix-matrix multiplcations, instead of **rank-one updates**.

Memory acces bandwidth

Recall that matrix-matrix multiplications have a much higher **computational intensity** (flops per memory access), hence they better exploit memory hierarchies and achieve higher performance on modern architectures.

- From the previous explanation, we change that each panel of A and B consist now of a block of columns/rows.

Blocking to further improve performance

We can reformulate the algorithm to use matrix-matrix multiplcations, instead of **rank-one updates**.

Memory acces bandwidth

Recall that matrix-matrix multiplications have a much higher **computational intensity** (flops per memory access), hence they better exploit memory hierarchies and achieve higher performance on modern architectures.

- From the previous explanation, we change that each panel of A and B consist now of a block of columns/rows.
- An **additional advantage of blocking** is that it reduces the number of messages incurred $\Rightarrow$ lower latency cost/communication overhead.

Some practical considerations

2 PBLAS and ScaLAPACK

- Choose nb to balance computation and communication.
- Square-ish grids minimize communication volume.
- Use optimized local BLAS for the inner GEMM.
- Synchronize only for timing, not for correctness.
- The routine is called via `PDGEMM`.
- A, B, and C are distributed with 2D block-cyclic layout.

```
CALL PDGEMM(TRANSA, TRANSB, M, N, K, ALPHA,
            A, IA, JA, DESC_A,
            B, IB, JB, DESC_B,
            BETA, C, IC, JC, DESC_C)
```

PDGEMM parameters


```
CALL PDGEMM(TRANSA, TRANSB, M, N, K, ALPHA,
            A, IA, JA, DESC_A,
            B, IB, JB, DESC_B,
            BETA, C, IC, JC, DESC_C)
```

- `M, N, K` define the sizes of C, A, and B.
- `IA, JA` and `IB, JB` select submatrices.
- `DESC_A`, `DESC_B`, `DESC_C` hold distribution metadata.
- `ALPHA` and `BETA` are scalars.

Creating descriptors for PDGEMM

Assume $A \in \mathbb{R}^{m \times k}$, $B \in \mathbb{R}^{k \times n}$, $C \in \mathbb{R}^{m \times n}$.

```
call descinit(descA, m, k, nb, nb, 0, 0, ictxt, lldA, info)
call descinit(descB, k, n, nb, nb, 0, 0, ictxt, lldB, info)
call descinit(descC, m, n, nb, nb, 0, 0, ictxt, lldC, info)
```

- Block sizes are typically identical for all matrices.
- `lld*` are the local leading dimensions.

Calling PDGEMM


```
call MPI_Barrier(MPI_COMM_WORLD, ierr)
t0 = MPI_Wtime()
do i = 1, nreps
  call pdgemm('N', 'N', m, n, k, alpha, A, 1, 1, descA, &
              B, 1, 1, descB, beta, C, 1, 1, descC)
end do
call MPI_Barrier(MPI_COMM_WORLD, ierr)
t1 = MPI_Wtime()
elapsed_time = t1 - t0
```

- Synchronize before and after to measure wall-clock time.
- `nreps` improves timing stability.

GFLOPS for PDGEMM

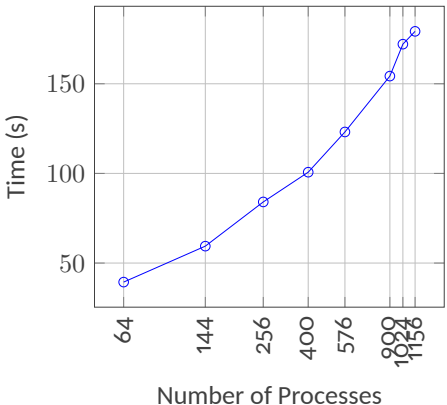
The operation count for GEMM is

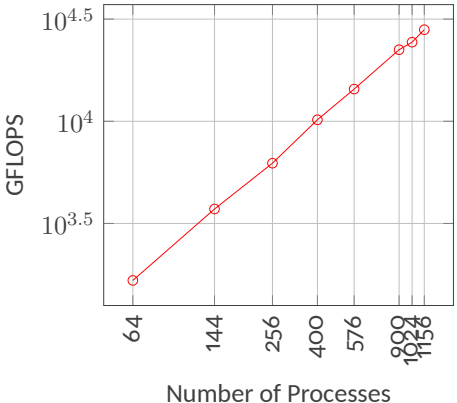
$$\text{FLOPS} = 2 \cdot m \cdot n \cdot k.$$

```
gflops = (2.0d0 * dble(m) * dble(n) * dble(k) * dble(nreps)) &
         / max_elapsed / 1.0d9
avg_time = max_elapsed / dble(nreps)
avg_gflops = (2.0d0 * dble(m) * dble(n) * dble(k)) / avg_time / 1.0d9
```

- Use the max time across processes for a conservative metric.

PDGEMM performance example





Conclusions, summary, and next steps

Today we have:

- Discussed the distributed matrix-vector multiplication and its scalability.
- Discussed the distributed matrix-matrix multiplication and its scalability.
- Presented practical aspects of using ScaLAPACK routines.

Next time:

- We will discuss more complex parallel algorithms for linear algebra.
- We will stat looking into GPU acceleration.