Chapter 5

Distributed Memory Systems: Part V

Data Dsitribution Schemes in Distributed LU

For a 2d data field (like a matrix) there are basically 3 types of data distribution patterns:

- **row/column blocks,**
- **row/column cyclic,**
- **checkerboard.**

All of them have their advantages and disadvantages in different algorithms. We will treat them all in the case of the LU decomposition in the following.

Data Dsitribution Schemes in Distributed LU

Multi-Collective Communication

Algorithm 7: Gaussian elimination – row-by-row-version

Input: $A \in \mathbb{R}^{n \times n}$ allowing LU decomposition
Output: A overwritten by L, U

```
1 for k = 1 : n − 1 do
2     A(k + 1 : n, k) = A(k + 1 : n, b)/A(k, k);
3     for i = k + 1 : n do
4         for j = k + 1 : n do
5             A(i, j) = A(i, j) − A(i, k)A(k, j);
```

Data Dsitribution Schemes in Distributed LU

Before diving into the details of data distribution we recall that after 4 steps of the row-by-row LU decomposition we have the following:

$$A^{(4)} = \begin{bmatrix} u_{1,1} & u_{1,2} & u_{1,3} & u_{1,4} & u_{1,5} & u_{1,6} & u_{1,7} & u_{1,8} & u_{1,9} & u_{1,10} \\ l_{2,1} & u_{2,2} & u_{2,3} & u_{2,4} & u_{2,5} & u_{2,6} & u_{2,7} & u_{2,8} & u_{2,9} & u_{2,10} \\ l_{3,1} & l_{3,2} & u_{3,3} & u_{3,4} & u_{3,5} & u_{3,6} & u_{3,7} & u_{3,8} & u_{3,9} & u_{3,10} \\ l_{4,1} & l_{4,2} & l_{4,3} & u_{4,4} & u_{4,5} & u_{4,6} & u_{4,7} & u_{4,8} & u_{4,9} & u_{4,10} \\ l_{5,1} & l_{5,2} & l_{5,3} & l_{5,4} & a_{5,5} & a_{5,6} & a_{5,7} & a_{5,8} & a_{5,9} & a_{5,10} \\ l_{6,1} & l_{6,2} & l_{6,3} & l_{6,4} & a_{6,5} & a_{6,6} & a_{6,7} & a_{6,8} & a_{6,9} & a_{6,10} \\ l_{7,1} & l_{7,2} & l_{7,3} & l_{7,4} & a_{7,5} & a_{7,6} & a_{7,7} & a_{7,8} & a_{7,9} & a_{7,10} \\ l_{8,1} & l_{8,2} & l_{8,3} & l_{8,4} & a_{8,5} & a_{8,6} & a_{8,7} & a_{8,8} & a_{8,9} & a_{8,10} \\ l_{9,1} & l_{9,2} & l_{9,3} & l_{9,4} & a_{9,5} & a_{9,6} & a_{9,7} & a_{9,8} & a_{9,9} & a_{9,10} \\ l_{10,1} & l_{10,2} & l_{10,3} & l_{10,4} & a_{10,5} & a_{10,6} & a_{10,7} & a_{10,8} & a_{10,9} & a_{10,10} \end{bmatrix}$$

Furthermore the blue and green parts will no longer be touched and the algorithm proceeds on the smaller lower right part $A(5:10, 5:10)$ only.

Row-/Column Block Distribution

Basic Idea:

Group the rows/columns in blocks of $\lceil \frac{n}{p} \rceil$. Each processor then works on one of those blocks, performing all necessary operations that treat any rows/columns in the scope.

$$A^{(4)} = \begin{bmatrix} u_{1,1} & u_{1,2} & u_{1,3} & u_{1,4} & u_{1,5} & u_{1,6} & u_{1,7} & u_{1,8} & u_{1,9} & u_{1,10} \\ l_{2,1} & u_{2,2} & u_{2,3} & u_{2,4} & u_{2,5} & u_{2,6} & u_{2,7} & u_{2,8} & u_{2,9} & u_{2,10} \\ l_{3,1} & l_{3,2} & u_{3,3} & u_{3,4} & u_{3,5} & u_{3,6} & u_{3,7} & u_{3,8} & u_{3,9} & u_{3,10} \\ l_{4,1} & l_{4,2} & l_{4,3} & u_{4,4} & u_{4,5} & u_{4,6} & u_{4,7} & u_{4,8} & u_{4,9} & u_{4,10} \\ l_{5,1} & l_{5,2} & l_{5,3} & l_{5,4} & a_{5,5} & a_{5,6} & a_{5,7} & a_{5,8} & a_{5,9} & a_{5,10} \\ l_{6,1} & l_{6,2} & l_{6,3} & l_{6,4} & a_{6,5} & a_{6,6} & a_{6,7} & a_{6,8} & a_{6,9} & a_{6,10} \\ l_{7,1} & l_{7,2} & l_{7,3} & l_{7,4} & a_{7,5} & a_{7,6} & a_{7,7} & a_{7,8} & a_{7,9} & a_{7,10} \\ l_{8,1} & l_{8,2} & l_{8,3} & l_{8,4} & a_{8,5} & a_{8,6} & a_{8,7} & a_{8,8} & a_{8,9} & a_{8,10} \\ l_{9,1} & l_{9,2} & l_{9,3} & l_{9,4} & a_{9,5} & a_{9,6} & a_{9,7} & a_{9,8} & a_{9,9} & a_{9,10} \\ l_{10,1} & l_{10,2} & l_{10,3} & l_{10,4} & a_{10,5} & a_{10,6} & a_{10,7} & a_{10,8} & a_{10,9} & a_{10,10} \end{bmatrix}$$

Row-/Column Block Distribution

Processors $\mathbf{P_1}$ and $\mathbf{P_2}$ have no more work do do after step 4. $\rightsquigarrow$ bad load balancing among the processors.

As a consequence we should not use the block distribution in cases when not the entire matrix is involved in all computations to make sure that all processors are equally well loaded. That means for parallel matrix-vector or matrix-matrix products it may serve well, but for the LU we need to find a data distribution that has a better distribution of the workload.

Cyclic-row/-column Distribution

Basic Idea:

Instead of distributing blocks of rows/columns assign a single row/column to a process until all got one and then start over until all rows/columns are distributed.

$$A^{(4)} = \begin{bmatrix} u_{1,1} & u_{1,2} & u_{1,3} & u_{1,4} & u_{1,5} & u_{1,6} & u_{1,7} & u_{1,8} & u_{1,9} & u_{1,10} \\ l_{2,1} & u_{2,2} & u_{2,3} & u_{2,4} & u_{2,5} & u_{2,6} & u_{2,7} & u_{2,8} & u_{2,9} & u_{2,10} \\ l_{3,1} & l_{3,2} & u_{3,3} & u_{3,4} & u_{3,5} & u_{3,6} & u_{3,7} & u_{3,8} & u_{3,9} & u_{3,10} \\ l_{4,1} & l_{4,2} & l_{4,3} & u_{4,4} & u_{4,5} & u_{4,6} & u_{4,7} & u_{4,8} & u_{4,9} & u_{4,10} \\ l_{5,1} & l_{5,2} & l_{5,3} & l_{5,4} & a_{5,5} & a_{5,6} & a_{5,7} & a_{5,8} & a_{5,9} & a_{5,10} \\ l_{6,1} & l_{6,2} & l_{6,3} & l_{6,4} & a_{6,5} & a_{6,6} & a_{6,7} & a_{6,8} & a_{6,9} & a_{6,10} \\ l_{7,1} & l_{7,2} & l_{7,3} & l_{7,4} & a_{7,5} & a_{7,6} & a_{7,7} & a_{7,8} & a_{7,9} & a_{7,10} \\ l_{8,1} & l_{8,2} & l_{8,3} & l_{8,4} & a_{8,5} & a_{8,6} & a_{8,7} & a_{8,8} & a_{8,9} & a_{8,10} \\ l_{9,1} & l_{9,2} & l_{9,3} & l_{9,4} & a_{9,5} & a_{9,6} & a_{9,7} & a_{9,8} & a_{9,9} & a_{9,10} \\ l_{10,1} & l_{10,2} & l_{10,3} & l_{10,4} & a_{10,5} & a_{10,6} & a_{10,7} & a_{10,8} & a_{10,9} & a_{10,10} \end{bmatrix}$$

Cyclic-row/-column Distribution

Obviously now the processors only start to become idle after $n - p$ steps of the outermost loop, i.e. in $A^{(n-p)}$, which is reasonable for $p \ll n$. Still basically every processor is responsible for $\lceil \frac{n}{p} \rceil$ rows.

Cyclic-row/-column Distribution

Obviously now the processors only start to become idle after $n - p$ steps of the outermost loop, i.e. in $A^{(n-p)}$, which is reasonable for $p \ll n$. Still basically every processor is responsible for $\lceil \frac{n}{p} \rceil$ rows.

Pivoting

Since pivoting adds a considerable amount of extra communication effort, we do not neglect it here in contrast to earlier appearances. However, we restrict ourselves to the case of column pivoting. That means as the first step of the outer for loop we add the pivot selection and row swapping.

Data Dsitribution Schemes in Distributed LU

Cyclic-row/-column Distribution

Algorithm 7: Gaussian elimination – row-by-row-version

Input: $A \in \mathbb{R}^{n \times n}$ allowing LU decomposition
Output: A overwritten by L, U

```
1 for k = 1 : n − 1 do
2     k_0 = argmax_{i=k:n} |A(i, k)|;
3     Swap rows k and k_0;
4     A(k + 1 : n, k) = A(k + 1 : n, b)/A(k, k);
5     for i = k + 1 : n do
6         for j = k + 1 : n do
7             A(i, j) = A(i, j) − A(i, k)A(k, j);
```

Cyclic-row/-column Distribution

Differences to the sequential case

1. **Determination of the pivot element.**

   The column below the diagonal is owned by several processors in the distributed parallel case. That means each processor finds its local pivot element and afterward they are compared among all processors.

Cyclic-row/-column Distribution

Differences to the sequential case

1. **Determination of the pivot element.**

   The column below the diagonal is owned by several processors in the distributed parallel case. That means each processor finds its local pivot element and afterward they are compared among all processors.

2. **Usage of the pivot element.**

   If we are lucky enough that the pivot row is owned by the same processor that owns the row containing the critical diagonal element, we are fine. We can perform a local row swap as in the sequential case. Otherwise the pivot row is exchanged with the process owning the "diagonal row".

Cyclic-row/-column Distribution

Differences to the sequential case

1. **Determination of the pivot element.**
   The column below the diagonal is owned by several processors in the distributed parallel case. That means each processor finds its local pivot element and afterward they are compared among all processors.
2. **Usage of the pivot element.**
   If we are lucky enough that the pivot row is owned by the same processor that owns the row containing the critical diagonal element, we are fine. We can perform a local row swap as in the sequential case. Otherwise the pivot row is exchanged with the process owning the "diagonal row".
3. **Distribution of the pivot row.**
   The pivot row is the key ingredient to the computation in the step. It is needed by all processors and thus needs to be broadcast to all active processors.

Cyclic-row/-column Distribution

Differences to the sequential case

1. **Determination of the pivot element.**
   The column below the diagonal is owned by several processors in the distributed parallel case. That means each processor finds its local pivot element and afterward they are compared among all processors.
2. **Usage of the pivot element.**
   If we are lucky enough that the pivot row is owned by the same processor that owns the row containing the critical diagonal element, we are fine. We can perform a local row swap as in the sequential case. Otherwise the pivot row is exchanged with the process owning the "diagonal row".
3. **Distribution of the pivot row.**
   The pivot row is the key ingredient to the computation in the step. It is needed by all processors and thus needs to be broadcast to all active processors.
4. **Computation of the matrix element updates.**
   The update step can now be performed as in the sequential case. Only, each processor just works through the local rows it owns.

Checkerboard Distribution

Basic Idea:

Distribution of the d-dimensional data array to a d-dimensional processor grid. Note that we can follow the blocked or cyclic variants just as in the case above.

$a_{1,1}$ $a_{1,2}$ **P_{11}** $a_{2,1}$ $a_{2,2}$	$a_{1,3}$ $a_{1,4}$ **P_{12}** $a_{2,3}$ $a_{2,4}$	$a_{1,5}$ $a_{1,6}$ **P_{13}** $a_{2,5}$ $a_{2,6}$	$a_{1,7}$ $a_{1,8}$ **P_{14}** $a_{2,7}$ $a_{2,8}$
$a_{3,1}$ $a_{3,2}$ **P_{21}** $a_{4,1}$ $a_{4,2}$	$a_{3,3}$ $a_{3,4}$ **P_{22}** $a_{4,3}$ $a_{4,4}$	$a_{3,5}$ $a_{3,6}$ **P_{23}** $a_{4,5}$ $a_{4,6}$	$a_{3,7}$ $a_{3,8}$ **P_{24}** $a_{4,7}$ $a_{4,8}$
$a_{5,1}$ $a_{5,2}$ **P_{31}** $a_{6,1}$ $a_{6,2}$	$a_{5,3}$ $a_{5,4}$ **P_{32}** $a_{6,3}$ $a_{6,4}$	$a_{5,5}$ $a_{5,6}$ **P_{33}** $a_{6,5}$ $a_{6,6}$	$a_{5,7}$ $a_{5,8}$ **P_{34}** $a_{6,7}$ $a_{6,8}$
$a_{7,1}$ $a_{7,2}$ **P_{41}** $a_{8,1}$ $a_{8,2}$	$a_{7,3}$ $a_{7,4}$ **P_{42}** $a_{8,3}$ $a_{8,4}$	$a_{7,5}$ $a_{7,6}$ **P_{43}** $a_{8,5}$ $a_{8,6}$	$a_{7,7}$ $a_{7,8}$ **P_{44}** $a_{8,7}$ $a_{8,8}$

(a) blocked distribution

$a_{1,1}$ $a_{1,5}$ **P_{11}** $a_{5,1}$ $a_{5,5}$	$a_{1,2}$ $a_{1,6}$ **P_{12}** $a_{5,2}$ $a_{5,6}$	$a_{1,3}$ $a_{1,7}$ **P_{13}** $a_{5,3}$ $a_{5,7}$	$a_{1,4}$ $a_{1,8}$ **P_{14}** $a_{5,4}$ $a_{5,8}$
$a_{2,1}$ $a_{2,5}$ **P_{21}** $a_{6,1}$ $a_{6,5}$	$a_{2,2}$ $a_{2,6}$ **P_{22}** $a_{6,2}$ $a_{6,6}$	$a_{2,3}$ $a_{2,7}$ **P_{23}** $a_{6,3}$ $a_{6,7}$	$a_{2,4}$ $a_{2,8}$ **P_{24}** $a_{6,4}$ $a_{6,8}$
$a_{3,1}$ $a_{3,5}$ **P_{31}** $a_{7,1}$ $a_{7,5}$	$a_{3,2}$ $a_{3,6}$ **P_{32}** $a_{7,2}$ $a_{7,6}$	$a_{3,3}$ $a_{3,7}$ **P_{33}** $a_{7,3}$ $a_{7,7}$	$a_{3,4}$ $a_{3,8}$ **P_{34}** $a_{7,4}$ $a_{7,8}$
$a_{4,1}$ $a_{4,5}$ **P_{41}** $a_{8,1}$ $a_{8,5}$	$a_{4,2}$ $a_{4,6}$ **P_{42}** $a_{8,2}$ $a_{8,6}$	$a_{4,3}$ $a_{4,7}$ **P_{43}** $a_{8,3}$ $a_{8,7}$	$a_{4,4}$ $a_{4,8}$ **P_{44}** $a_{8,4}$ $a_{8,8}$

(b) cyclic distribution

Checkerboard Distribution

Definition

Let $n = \prod_{i=1}^{d} n_i$ be the total problem size and n_i the degrees of freedom in the i-th direction. Also p, as before, the number of processors in total. We call $\mathbf{p} = (p_1, \ldots, p_d)$ a processor distribution if it holds

$$p \leq \prod_{i=1}^{d} p_i.$$

On each processor we assume a local data distribution $\mathbf{b} = (b_1, \ldots, b_d)$ with

$$n \leq \prod_{i=1}^{d} p_i b_i.$$

Ideally we want to have equality in both cases to achieve optimal load balancing.

An Alternative for the LU Using Distributed BLAS and LAPACK

The PBLAS project (details later) aims at providing a parallel distributed version of the BLAS library. In the previous Chapters we have investigated level 3 BLAS based block outer product versions of the LU decomposition.

Data Distribution for other Problems

domain decomposition

Similar to the splitting of the matrix into blocks on which smaller subproblems are solved, in domain decomposition[a] methods for boundary value problems the objective domain on which the problem is to be solved is subdivided into smaller parts. Then on each part a smaller independent boundary value problem is solved. The interaction between subdomains is only necessary if their intersection is non empty, i.e., they have a common "boundary", the interface. In each iteration step both processes rely on the result of the prior step and exchange the data on the interface to make it fit in a post-processing procedure.

[a] http://www.ddm.org

- The interface is sometimes also called halo.
- The interface may be a single layer of unknowns, but can also be extended. One then speaks of overlapping domain decomposition methods.

Relevant Software and Libraries

Implementations of the MPI Standard

- Open MPI, Current feature release 1.8.6 implements MPI-3[a]
- MPICH 3.2b3 (version 3.2 preview release of June 4, 2015) supports MPI-3.1[b]
- MVAPICH: The current MVAPICH2 2.1 is based on MPICH 3.1.4 [c]
- Intel® MPI Library: version 5.0 implements MPI-3.0 [d]

---

[a] `http://www.openmpi.org`
[b] `http://www.mpich.org/`
[c] `http://mvapich.cse.ohio-state.edu/`
[d] `http://software.intel.com/en-us/intel-mpi-library`

Relevant Software and Libraries

Scientific Software

- BLACS (Basic Linear Algebra Communication Subprograms) "is an ongoing investigation whose purpose is to create a linear algebra oriented message passing interface that may be implemented efficiently and uniformly across a large range of distributed memory platforms." [a]
- ScaLAPACK a BLACS-based scalable distributed implementation of LAPACK (current version 2.0.2 of May 1, 2012) [b]
- PBLAS (Parallel Basic Linear Algebra Subprograms) subproject of the above [c]
- Boost starting with version 1.35 has a boost.MPI module providing a C++ friendly MPI framework. [d]

---

[a] http://www.netlib.org/blacs/
[b] http://www.netlib.org/scalapack/
[c] http://www.netlib.org/scalapack/pblas_qref.html
[d] http://www.boost.org/

Relevant Software and Libraries

Scientific Software

- PETSC "*is a suite of data structures and routines for the scalable (parallel) solution of scientific applications modeled by partial differential equations.*" [a]
- SLEPC is the **S**calable **L**ibrary for **E**igenvalue **P**roblem **C**omputations.[b]
- PARPACK an extension to the ARPACK for eigenvalue computations using MPI and BLACS for parallel execution.[c]

---

[a] `http://www.mcs.anl.gov/petsc/`
[b] `http://www.grycap.upv.es/slepc/`
[c] `http://www.caam.rice.edu/software/ARPACK/`