Distributed and Parallel Dense Block Linear Algebra using YML and XMP

Jérôme Gurhem

CNRS/Maison de la Simulation Saclay, Université Lille 1

5^{th} April 2017

Serge Petiton, CNRS/Maison de la Simulation Saclay, Université Lille 1
Miwako Tsuji, RIKEN Advanced Institute for Computational Science

Outline

Introduction

- Exascale is coming soon in Japan.
- Need new paradigms of programming.
- YML/XMP is a 2 levels paradigm.
- LU factorization developed with those languages.
- Experiments on K.

YML

- Programming and executing environment for HPC, Cloud, P2P and Grid.
- Parallelism expression with a graph language (YvetteML).
- Independent from systems and middle-wares.
- Designing applications by components. $\Rightarrow$ re-usability
- Components implemented with XMP.

LU factorization

- A is a dense matrix per blocks.
- L is a lower triangular matrix per blocks with diagonals blocks equal to the identity.
- U is an upper triangular matrix per blocks.

- We want to find L and U as $A = L \cdot U$.

$$A = \begin{bmatrix} I & 0 & 0 & 0 \\ L_{2,1} & I & 0 & 0 \\ L_{3,1} & L_{3,2} & I & 0 \\ L_{4,1} & L_{4,2} & L_{4,3} & I \end{bmatrix} \cdot \begin{bmatrix} U_{1,1} & U_{1,2} & U_{1,3} & U_{1,4} \\ 0 & U_{2,2} & U_{2,3} & U_{2,4} \\ 0 & 0 & U_{3,3} & U_{3,4} \\ 0 & 0 & 0 & U_{4,4} \end{bmatrix}$$

- A, L and U have p by p blocks of n by n values.

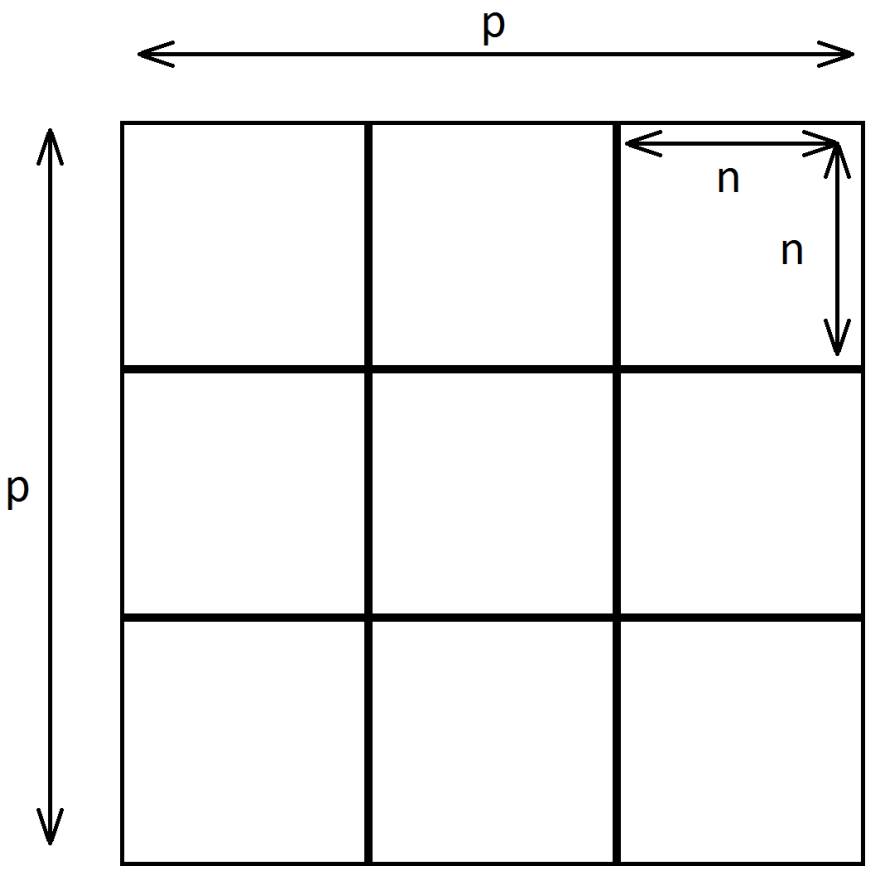


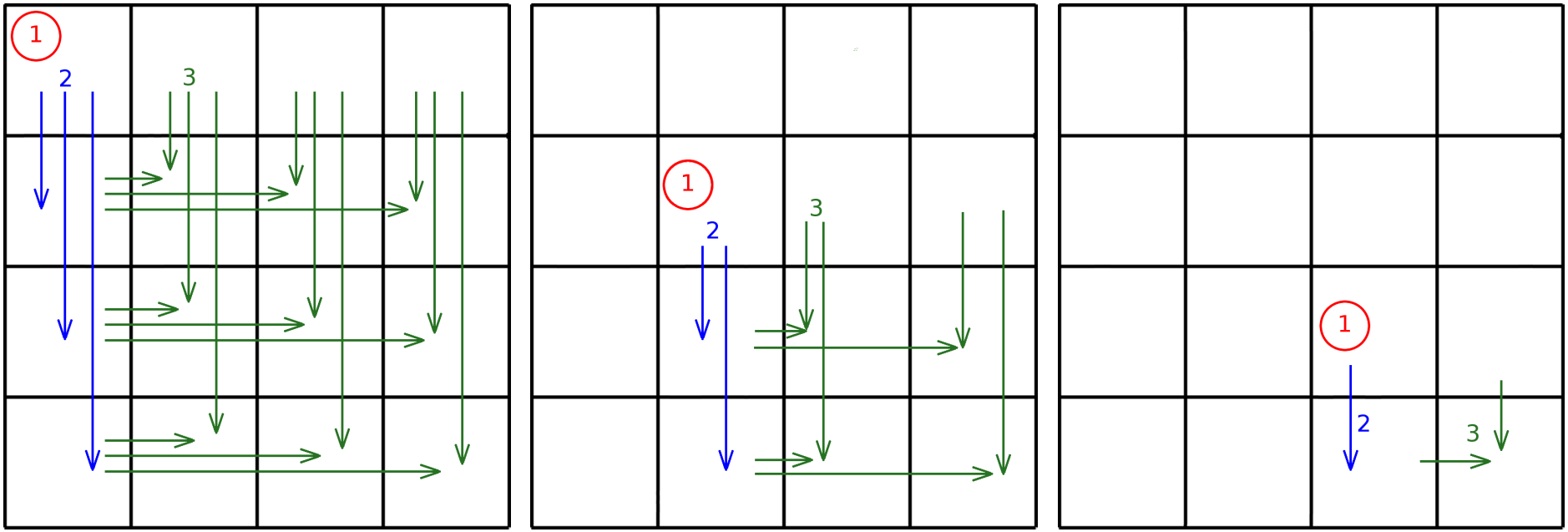
Figure 1: Shape of a block matrix

- We will transform A into L and U as the same matrix.
- $$\begin{bmatrix} A_{1,1} & A_{1,2} & A_{1,3} & A_{1,4} \\ A_{2,1} & A_{2,2} & A_{2,3} & A_{2,4} \\ A_{3,1} & A_{3,2} & A_{3,3} & A_{3,4} \\ A_{4,1} & A_{4,2} & A_{4,3} & A_{4,4} \end{bmatrix} \Rightarrow \begin{bmatrix} U_{1,1} & U_{1,2} & U_{1,3} & U_{1,4} \\ L_{2,1} & U_{2,2} & U_{2,3} & U_{2,4} \\ L_{3,1} & L_{3,2} & U_{3,3} & U_{3,4} \\ L_{4,1} & L_{4,2} & L_{4,3} & U_{4,4} \end{bmatrix}$$

for k **from** *1* **to** *p-1* **do**
  (1) compute $[A_{k,k}^{(k)}]^{-1}$ — *inverse*
  for i **from** $k+1$ **to** p **do**
    (2) $A_{i,k}^{(k+1)} = A_{i,k}^{(k)} \cdot [A_{k,k}^{(k)}]^{-1}$ — *prodMat*
    for j **from** $k+1$ **to** p **do**
      (3) $A_{i,j}^{(k+1)} = A_{i,j}^{(k)} - A_{i,k}^{(k+1)} \cdot A_{k,j}^{(k)})$ — *prodDiff*
    end for
  end for
end for

Algorithm 1: LU Factorization

Figure 2: LU factorization of a 4×4 block matrix



YML implementation

- But we can go further by expressing dependencies between each tasks then scheduling everything.
- Dependencies can be expressed using a graph of tasks as YvetteML.
- Tasks will be computations on blocks.

YvetteML graph for LU factorization

```
par(i:=0;blockcount-1)(j:=0;blockcount-1)
do
    if (j gt i) then
        par(k:=i+1;blockcount-1)
        do
            wait(p[k][j][i] and p[k][i][i+1] and p[i][j][i]);
            #A[k,j] = A[k,j] - A[k,i] * A[i,j]
            compute prodDiff(A[k][i],A[i][j],A[k][j],blocksize);
            notify(p[k][j][i+1]);
        enddo
    else
        if (i eq j) then
            if(i neq blockcount-1) then
                wait(p[i][i][i]);
                #B[i]=inverse(A[i,i])
                compute inversion(A[i][i],B[i],blocksize);
                notify(p[i][i][i+1]);
            endif
        else
            wait(p[j][j][j+1] and p[i][j][j]);
            #A[i,j] = A[i,j] * B[j]
            compute prodMat(A[i][j],B[j],blocksize);
            notify(p[i][j][j+1]);
        endif
    endif
enddo
```

Resolution of Linear System using a LU factorization

- Now we have a LU factorization of A so we can solve a linear system $Ax = b$ where b and x are vector per blocks.
- The system can be expressed as : $LUx = b$.
- There is two systems to solve $Ly = b$ then $Ux = y$.

$Ly = b$

- $Ly = b \Leftrightarrow \begin{cases} b_1 = y_1 \\ b_2 = L_{2,1} \cdot y_1 + y_2 \\ b_3 = L_{3,1} \cdot y_1 + L_{3,2} \cdot y_2 + y_3 \\ b_4 = L_{4,1} \cdot y_1 + L_{4,2} \cdot y_2 + L_{4,3} \cdot y_3 + y_4 \end{cases}$

- ```
  par(i:= 0;blockcount-2)
  do
      par(k:= i+1;blockcount-1)
      do
          wait(p[k][i][i+1] and lsystem[k][i] and lsystem[i][i]);

          #b[k] = b[k] - A[k,i] b[i]
          compute prodDiffMV(A[k][i],b[i],b[k],blocksize);

          notify(lsystem[k][i+1]);
      enddo
  enddo
  ```
  ```

$Ux = y$

- $$Ux = y \Rightarrow \begin{cases} y_1 = U_{1,4} \cdot x_4 + U_{1,3} \cdot x_3 + U_{1,2} \cdot x_2 + U_{1,1} \cdot x_1 \\ y_2 = U_{2,4} \cdot x_4 + U_{2,3} \cdot x_3 + U_{2,2} \cdot x_2 \\ y_3 = U_{3,4} \cdot x_4 + U_{3,3} \cdot x_3 \\ y_4 = U_{4,4} \cdot x_4 \end{cases}$$
- **for** j **from** p **to** *1* **step** *-1* **do**
  - solve $y_j = U_{j,j} \cdot x_j$ — *solveLinearSystem*
  - **for** i **from** *j-1* **to** *1* **step** *-1* **do**
    - $y_i = y_i - U_{i,j} \cdot x_j$ — *prodDiffMV*
  - **end for**
- **end for**

Algorithm 2: $Ux = y$

YML header of the component solveLinearSystem

```
<?xml version="1.0"?>
<component type="impl" name="XMP_solveLinearSystem"
   abstract="XMP_solveLinearSystem"
   description="solve Ax=b and save the results into b">
<impl lang="XMP"  nodes="CPU:(4)">
  <templates>
    <template name="t" format="block" size="64"/>
  </templates>
  <distribute>
    <param template="t" name="A0(64,64)" align="[i][*]:(i)"/>
    <param template="t" name="b0(64)" align="[i]:(i)"/>
  </distribute>
<header>
```

XMP implementation of solveLinearSystem

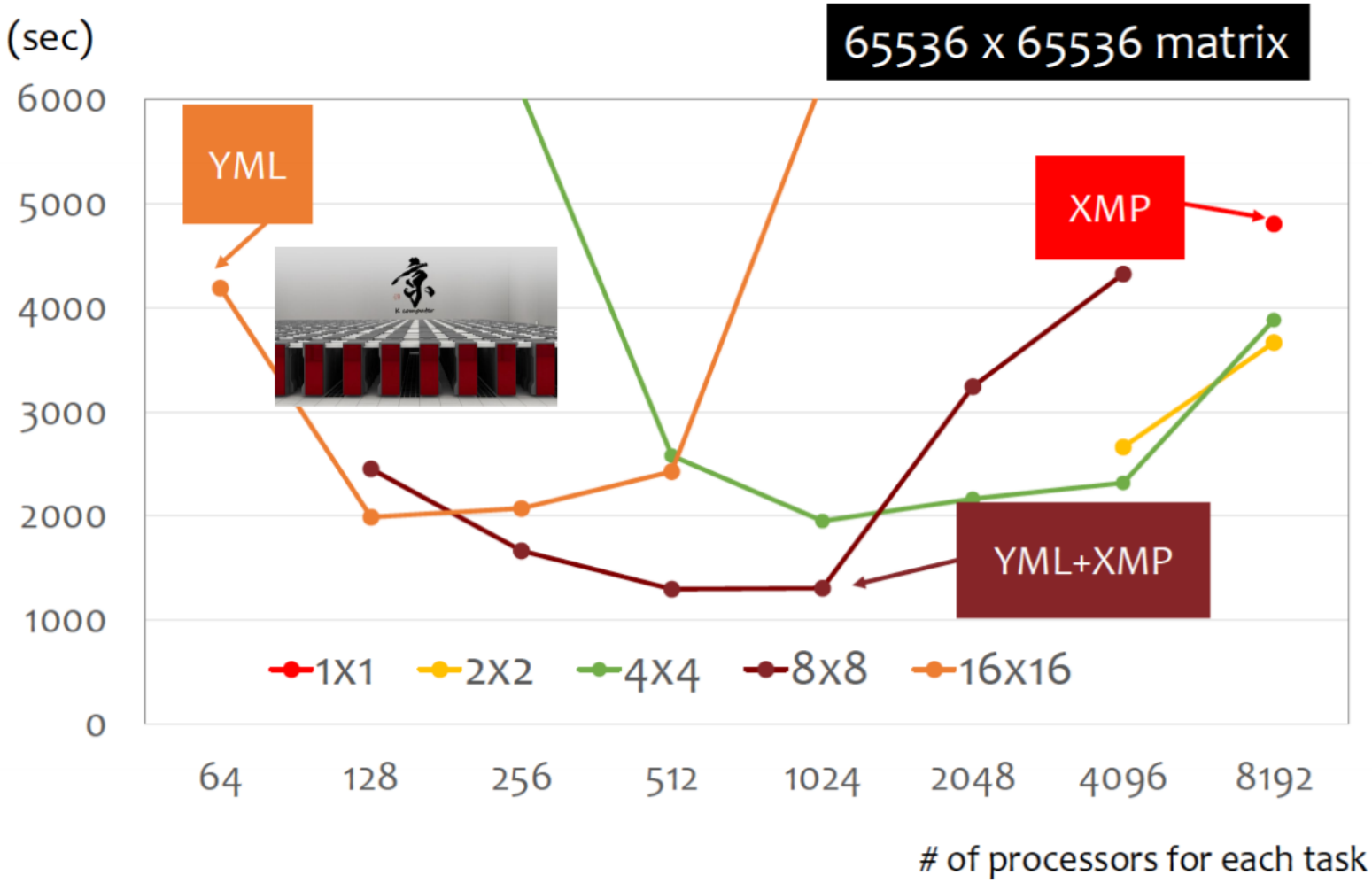
```
for(k=0;k<n-1;k++){
    btemp=0.0;
#pragma xmp task on t(k)
    {
        for(i=k+1;i<n;i++){
            A0[k][i] = A0[k][i] / A0[k][k];
        }
        b0[k] = b0[k] / A0[k][k];
        btemp = b0[k];
    }
#pragma xmp reduction(+:btemp)

    for(j=k+1;j<n;j++){

/*start spread */
        akj=0.0;
#pragma xmp task on t(k)
        akj = A0[k][j];
#pragma xmp reduction(+:akj)
/*end spread */

#pragma xmp loop (i) on t(i)
        for(i=k+1;i<n;i++){
            A0[i][j] = A0[i][j] - A0[i][k] * akj;
        }
```

Previous results of Gauss Jordan for inversion



Results from M.Tsuji, S.Petiton and M.Sato.

Experiments

One YML worker manages one task.

matrix size : 65 536 × 65 536
#procs = {2k,4k,8k,16k}

blocks	#procs/tasks	tasks
4×4	256	9

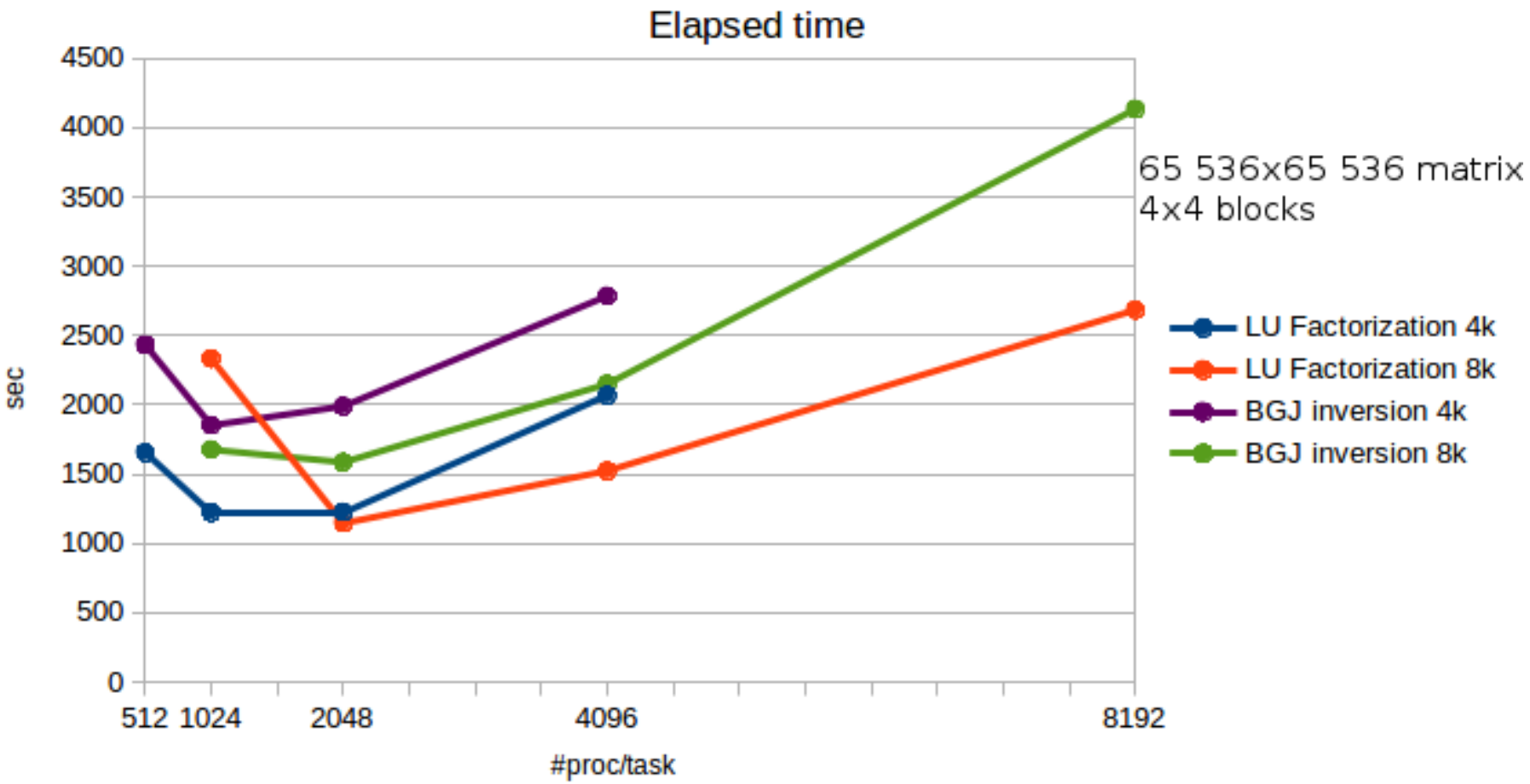
#procs=4k

blocks	#procs/tasks	tasks
4×4	512	8
4×4	1024	4
4×4	2048	2
4×4	4096	1

#procs=8k

blocks	#procs/tasks
4×4	1024
4×4	2048
4×4	4096
4×4	8192

Results


Elapsed time
4500
4000
3500
3000
2500
2000
1500
1000
500
0
sec
512 1024
2048
4096
8192
#proc/task
65 536x65 536 matrix
4x4 blocks
LU Factorization 4k
LU Factorization 8k
BGJ inversion 4k
BGJ inversion 8k

Conclusion

- Only a part of the results from LU factorization.
- Comparison between different methods to solve linear systems:
  - LU factorisation + forward and backward substitution
  - Gaussian elimination + backward substitution
  - Gauss Jordan for system resolution

Thank you for your attention.

Part of the results is obtained by using the K computer at the RIKEN Advanced Institute for Computational Science.