Data Structures to Implement the Sparse Vector in Crout ILU Preconditioner

Sparse Days 2019

11th July 2019

Raju Ram

PhD Student
CC-HPC
Fraunhofer ITWM
Fraunhofer-Platz 1
67663 Kaiserslautern
Germany

Data Structures to Implement the Sparse Vector in Crout ILU Preconditioner

- Iterative Solvers for Sparse Linear Systems
  - GaspiLS Linear Solver Library
- Crout Incomplete *LU* (*ILU*) factorization of *A*
  - Serial Crout ILU Method
  - Multilevel Crout ILU Method
  - Parallelization of Crout ILU Method
- Sparse Vector Update in Serial Crout ILU Factorization
  - Data Structures to Implement Sparse Vector
  - Profiler Readings

Sparse Iterative Linear Solvers

- Linear Solvers on Sparse Matrices are the base of Simulations in Science, Engineering.
- Solve for x in $Ax = b$.
- Preconditioning improves the convergence of linear solver since it reduces the condition number of the matrix.
- Typical kernel in Parallel Iterative Solvers:
  - Preconditioning step (line 4)
  - Sparse matrix vector multiplication (line 12)
  - Global dot product (line 8, 12)
  - Axpy operation (line 10, 14, 15)
- Preconditioning Step is most challenging kernel for scalability in parallel iterative solvers.

Algorithm 1 Preconditioned Conjugate Gradient (PCG) Algorithm

1: **procedure** $\mathrm{PCG}(A, b, x^{(0)}, tol, k_{max}, M)$
2: $r^{(0)} = f - Au^{(0)}, k = 1$ ▷ Initialization
3: **while** $(k < k_{max}$ **and** $\|r^{(k-1)}\| > tol)$ **do**
4: $z^{(k-1)} = M^{-1} r^{(k-1)}$ ▷ Preconditioning
5: **if** $k = 1$ **then**
6: $p^{(1)} = z^{(0)}$
7: **else**
8: $\beta_k = \frac{(r^{(k-1)})^T z^{(k-1)}}{(r^{(k-2)})^T z^{(k-2)}}$
9:
10: $p^{(k)} = z^{(k-1)} + \beta_k p^{(k-1)}$ ▷ Search direction
11: **end if**
12: $\alpha_k = \frac{(r^{(k-1)})^T z^{(k-1)}}{(p^{(k)})^T A p^{(k)}}$
13:
14: $x^{(k)} = x^{(k-1)} + \alpha_k p^{(k)}$ ▷ Iterate update
15: $r^{(k)} = r^{(k-1)} - \alpha_k A p^{(k)}$ ▷ Residual update
16: $k = k + 1$
17: **end while**
18: **return** $x^{(k-1)}$ ▷ The converged solution
19: **end procedure**

Global Address Space Programming Interface (GASPI)

- Asynchronous, single sided communication based standardized API.
- Developed keeping scalability in mind.
- Applications are developed in machine learning, big data, seismic imaging based on GASPI.
- We are building a scalable linear solver library based on GASPI to solve complex simulations in science and engineering.

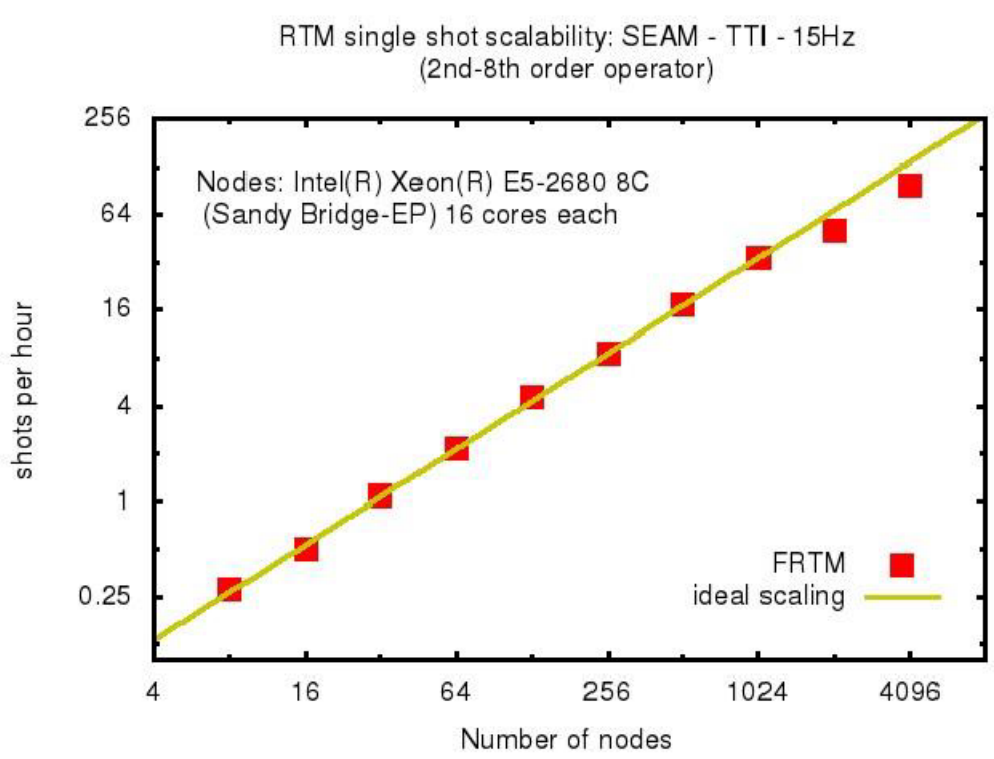
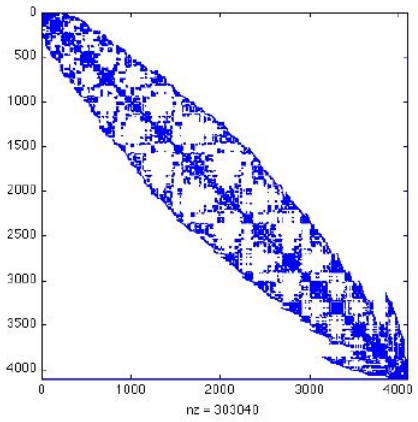


Abbildung: Ideal scalability until 1024 nodes [4]

GaspiLS linear solver library

- **Gaspi** based C++ **L**inear **S**olver Library
- Iterative solvers: (P)CG, BiPCGStab, GMRES
- Preconditioners: Jacobi, ILU, ILUM
- MPI interoperable
- Industry proven in CFD and FEM simulations
- OpenSource GPLv3 license library [1]



Fraunhofer ITWM

My research motivation

- Solving for z in $Mz = r$ with Incomplete LU Preconditioner (ILU) consists of two steps:

$$A = \begin{pmatrix} B & F \\ E & C \end{pmatrix} = LDU + E \approx$$  , where L, U are unit diagonal.

$LDUz = r$, solve for y in $Ly = r$ using forward substitution where $y = DUz$.

$DUz = y$, solve for z in $Uz = D^{-1}y$ using backward substitution.

- LDU Factorization and forward and backward substitution algorithm are serial in nature, thus **difficult to parallelize**.
- We want to decompose the problem into several tasks and run them in parallel as much as possible.

My research motivation

- Solving for z in $Mz = r$ with Incomplete LU Preconditioner (ILU) consists of two steps:

$$A = \begin{pmatrix} B & F \\ E & C \end{pmatrix} = LDU + E \approx$$  , where L, U are unit diagonal.

$LDUz = r$, solve for y in $Ly = r$ using forward substitution where $y = DUz$.

$DUz = y$, solve for z in $Uz = D^{-1}y$ using backward substitution.

- LDU Factorization and forward and backward substitution algorithm are serial in nature, thus **difficult to parallelize**.
- We want to decompose the problem into several tasks and run them in parallel as much as possible.

My research motivation

- Solving for z in $Mz = r$ with Incomplete LU Preconditioner (ILU) consists of two steps:

$$A = \begin{pmatrix} B & F \\ E & C \end{pmatrix} = LDU + E \approx$$  , where L, U are unit diagonal.

$LDUz = r$, solve for y in $Ly = r$ using forward substitution where $y = DUz$.

$DUz = y$, solve for z in $Uz = D^{-1}y$ using backward substitution.

- LDU Factorization and forward and backward substitution algorithm are serial in nature, thus **difficult to parallelize**.
- We want to decompose the problem into several tasks and run them in parallel as much as possible.

Data Structures to Implement the Sparse Vector in Crout ILU Preconditioner

- Iterative Solvers for Sparse Linear Systems
- Crout Incomplete *LU* (*ILU*) factorization of *A*
  - Serial Crout ILU Method
  - Multilevel Crout ILU Method
  - Parallelization of Crout ILU Method
- Sparse Vector Update in Serial Crout ILU Factorization

Incomplete LU (ILU) Preconditioner

- The iterative methods frequently incorporate incomplete LU (ILU) preconditioner because of its robustness, accuracy, and usability as a black-box preconditioner.
- The ILU Preconditioner is serial in nature.
- To introduce parallelism, Crout version of ILU is used as a preconditioner.
- Crout ILU may control the growth of error in the preconditioning operation by setting a bound on the inverse of the triangular (L and U) factors.

Incomplete LU (ILU) Preconditioner

- The iterative methods frequently incorporate incomplete LU (ILU) preconditioner because of its robustness, accuracy, and usability as a black-box preconditioner.
- The ILU Preconditioner is serial in nature.
- To introduce parallelism, Crout version of ILU is used as a preconditioner.
- Crout ILU may control the growth of error in the preconditioning operation by setting a bound on the inverse of the triangular (L and U) factors.

Incomplete LU (ILU) Preconditioner

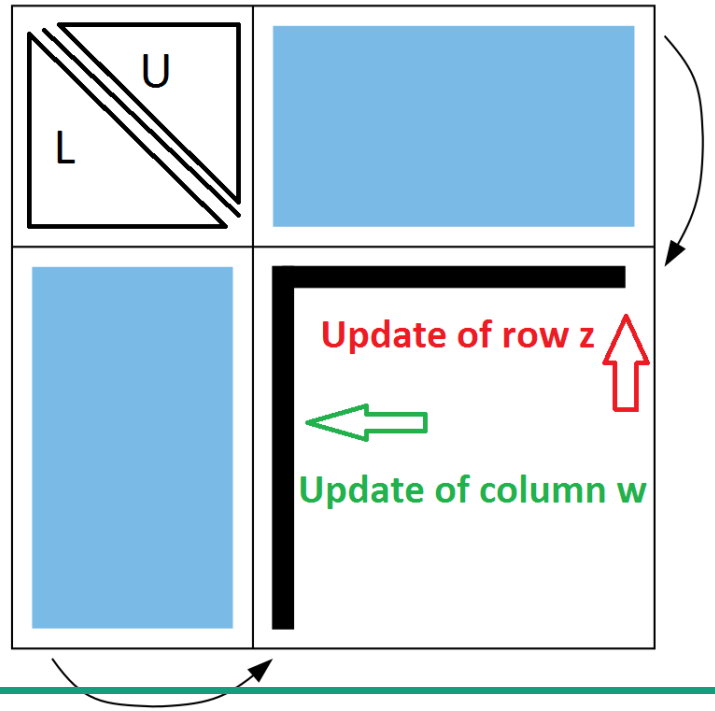
- The iterative methods frequently incorporate incomplete LU (ILU) preconditioner because of its robustness, accuracy, and usability as a black-box preconditioner.
- The ILU Preconditioner is serial in nature.
- To introduce parallelism, Crout version of ILU is used as a preconditioner.
- Crout ILU may control the growth of error in the preconditioning operation by setting a bound on the inverse of the triangular (L and U) factors.

Incomplete LU (ILU) Preconditioner

- The iterative methods frequently incorporate incomplete LU (ILU) preconditioner because of its robustness, accuracy, and usability as a black-box preconditioner.
- The ILU Preconditioner is serial in nature.
- To introduce parallelism, Crout version of ILU is used as a preconditioner.
- Crout ILU may control the growth of error in the preconditioning operation by setting a bound on the inverse of the triangular (L and U) factors.

Crout ILU Algorithm

- Until step $(k-1)$, LDU factorization of $A(1:k-1, 1:k-1)$ is computed.
- At kth step, row $z(\ A(k, k:n))$ and column $w(\ A(k+1:n, k))$ is updated.
- U is stored in CSR and L is stored in CSC format.



Algorithm 2 Crout version of ILU Algorithm: ILUC

1: **for** $k = 1 : n$ **do**
2: $z_{1:k-1} = 0, \quad z_{k:n} = a_{k,k:n}$ ▷ Initialize row z
3: **for** $(1 \le i \le k-1$ **and** $l_{ki} \neq 0)$ **do**
4: $z_{k:n} = z_{k:n} - l_{ki} * u_{i,k:n}$ ▷ Preparation to update $u_{k,:}$
5: **end do**
6: $w_{1:k} = 0, \quad w_{k+1:n} = a_{k+1:n,k}$ ▷ Initialize column z
7: **for** $(1 \le i \le k-1$ **and** $u_{ik} \neq 0)$ **do**
8: $w_{k+1:n} = w_{k+1:n} - u_{ik} * l_{k+1:n,i}$ ▷ Preparation to update $l_{:,k}$
9: **end do**
10: Apply a dropping rule to row z
11: Apply a dropping rule to column w
12: $u_{k,:} = z$
13: $l_{:,k} = w/u_{kk}, \quad l_{kk} = 1$
14: **end do**

Dropping entries in L and U

- Why dropping? : The complete LU decomposition of sparse matrix A introduce additional fill-ins in L and U. This leads to computation overhead during solution phase.

- Dropping strategies in $ILU(\tau, p)$:
  - τ : Drop an element whose magnitude is less than a tolerance τ.
  - p : Only keep p largest elements in magnitude in the k-th column of L and kth row of U.
    Drop all other elements.

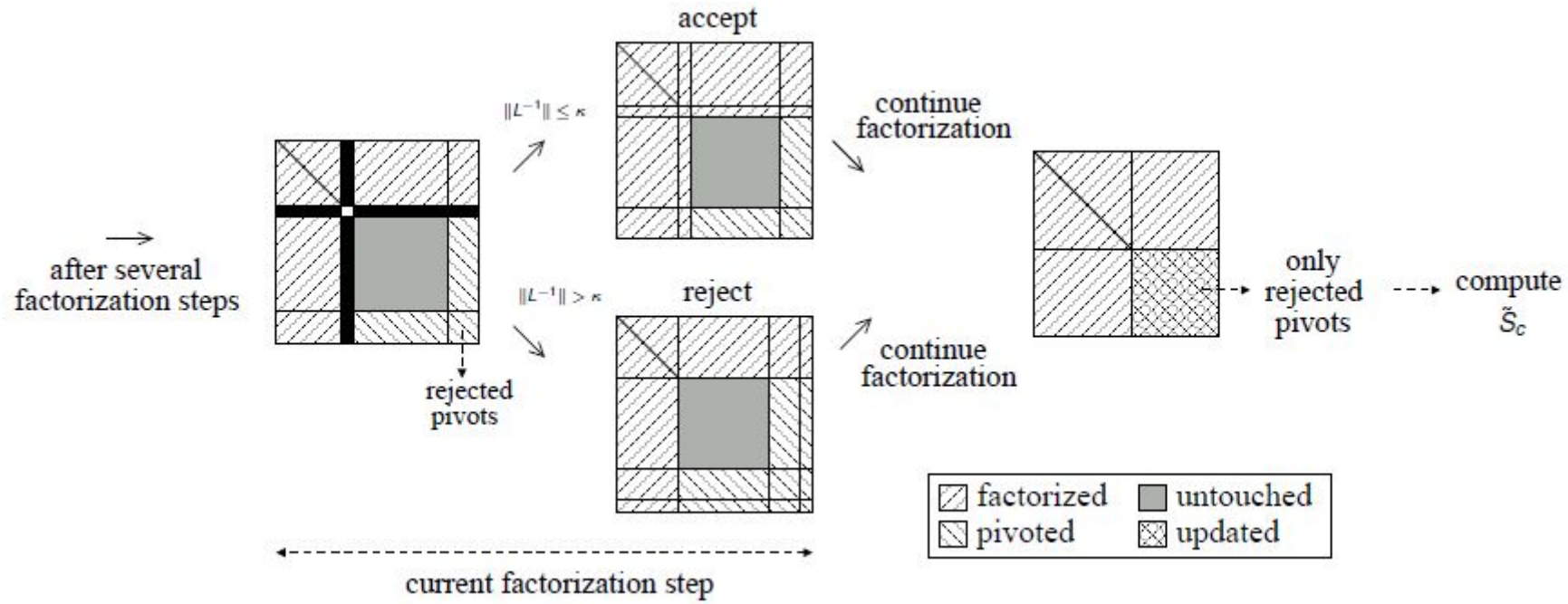
Dropping entries in L and U

- Why dropping? : The complete LU decomposition of sparse matrix A introduce additional fill-ins in L and U. This leads to computation overhead during solution phase.

- Dropping strategies in $ILU(\tau, p)$:
  - τ : Drop an element whose magnitude is less than a tolerance τ.
  - p : Only keep p largest elements in magnitude in the k-th column of L and kth row of U.
    Drop all other elements.

Inverse Based Pivoting in ILU

- Applying Crout ILU on A, we obtain $A = LDU + R$. The matrix R is generated due to dropping.
- Preconditioning matrix: $L^{-1}AU^{-1} = D + L^{-1}RU^{-1}$.
- Error matrix $L^{-1}RU^{-1}$ grows when $\|L^{-1}\|$ and $\|U^{-1}\|$ grows.
- If $\|L^{-1}\| \leq \kappa$ perform factorization, else move the pivot a_{kk} at the end of matrix.



(Courtesy: figure from [3]) José Ignacio Aliaga, Exploiting task and data parallelism in ILUPA preconditioned CG solver on NUMA architectures and many-core accelerators

Schur Complement Matrix Computation

- To bound the inverse norm, the matrix is not completely factorized at this stage. Instead, Partial Crout ILU factorization of A such that $B = L_B D_B U_B$.

$$A = \begin{bmatrix} B & E \\ F & C \end{bmatrix} = \begin{bmatrix} L_B & 0 \\ L_E & I \end{bmatrix} \begin{bmatrix} D_B & 0 \\ 0 & S_C \end{bmatrix} \begin{bmatrix} U_B & U_F \\ 0 & I \end{bmatrix} + E$$

- Schur complment matrix S_C is computed that will be factorized at next level.

$$S_C \approx C - L_E D_B U_F.$$

Schur Complement Matrix Computation

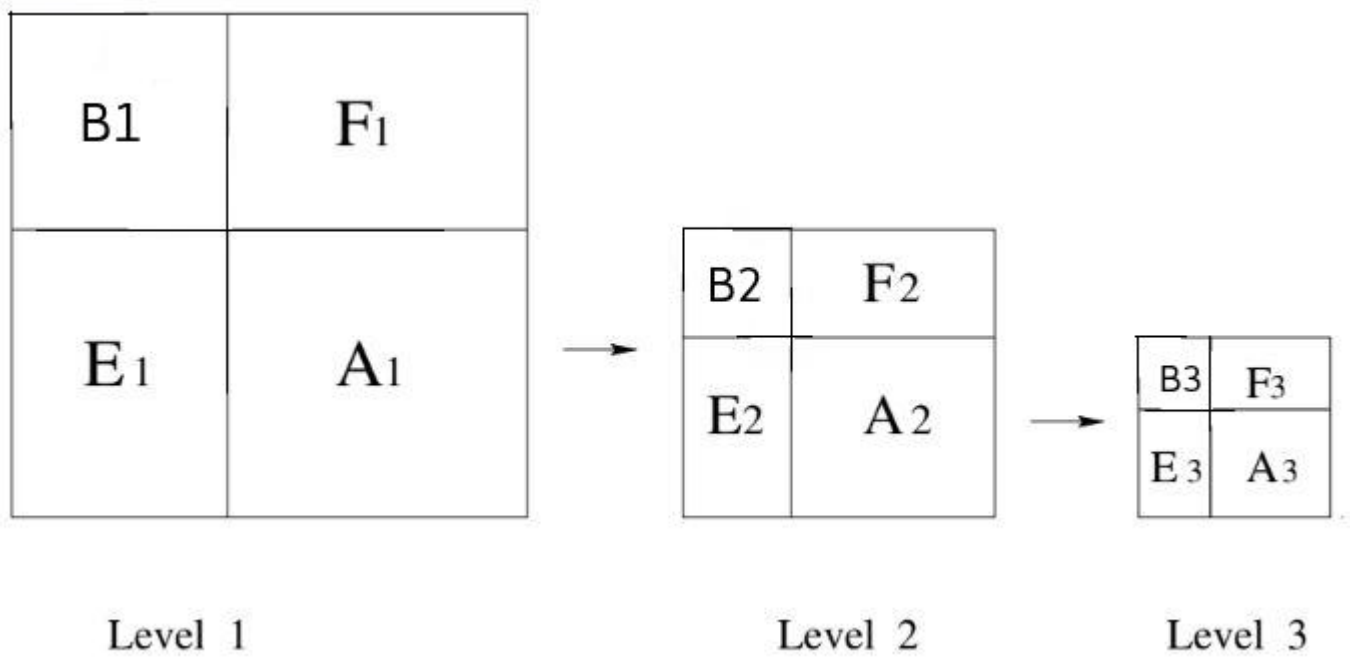
- To bound the inverse norm, the matrix is not completely factorized at this stage. Instead, Partial Crout ILU factorization of A such that $B = L_B D_B U_B$.

$$A = \begin{bmatrix} B & E \\ F & C \end{bmatrix} = \begin{bmatrix} L_B & 0 \\ L_E & I \end{bmatrix} \begin{bmatrix} D_B & 0 \\ 0 & S_C \end{bmatrix} \begin{bmatrix} U_B & U_F \\ 0 & I \end{bmatrix} + E$$

- Schur complment matrix S_C is computed that will be factorized at next level.

$$S_C \approx C - L_E D_B U_F.$$

Multilevel Crout ILU Method



Algorithm 3 Multilevel Crout Algorithm

```
1:  procedure MultiLevel_Crout_ILU(A)
2:      if (coarsest level) then
3:          Decompose Crout LDU factorization.
4:          return
5:      else
6:          Compute Crout LDU formulation of submatrix n_B × n_B of A
7:          Compute the Schur complement matrix S_C
8:          Call MULTILEVEL_CROUT_ILU (S_C).
9:          return
10:     end if
11: end procedure
```

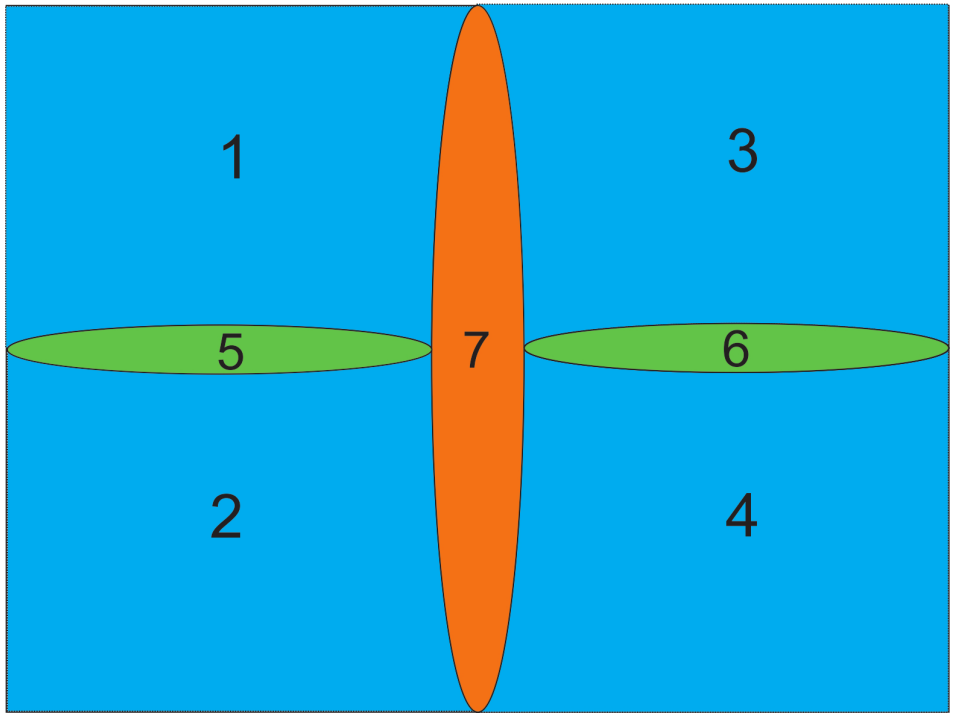

Fraunhofer ITWM

Data Structures to Implement the Sparse Vector in Crout ILU Preconditioner

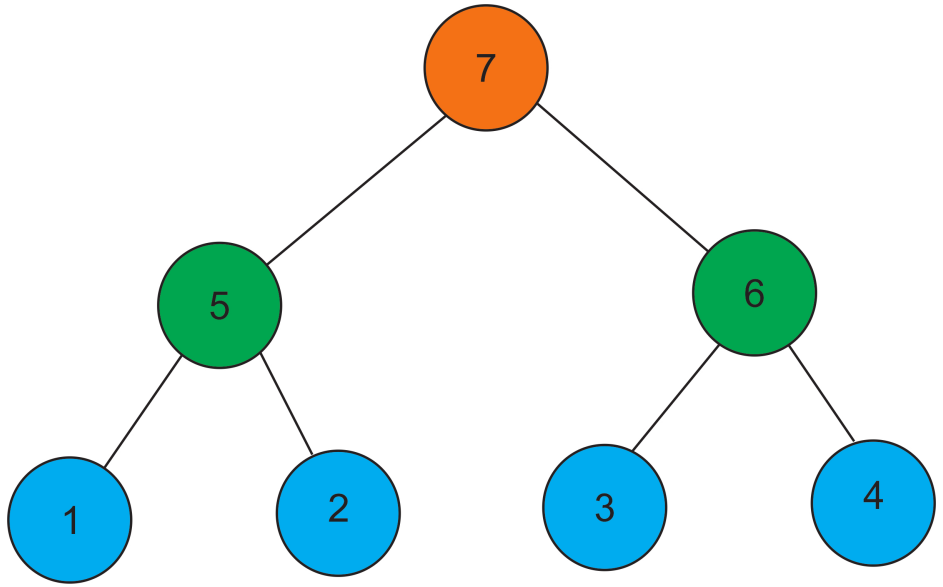
- Iterative Solvers for Sparse Linear Systems
- Crout Incomplete *LU* (*ILU*) factorization of *A*
  - Serial Crout ILU Method
  - Multilevel Crout ILU Method
  - Parallelization of Crout ILU Method
- Sparse Vector Update in Serial Crout ILU Factorization

How to extract Parallelism in Crout LU decomposition

- Sparse matrix A can be represented in an undirected graph G_A.
- Nested Dissection of G_A leads to a binary task tree.



(a) The graph G_A partitioned using 2 level nested dissection



(b) Task dependency tree corresponding to a)

Abbildung: Extracting task level parallelism from ILU algorithm

Fraunhofer ITWM

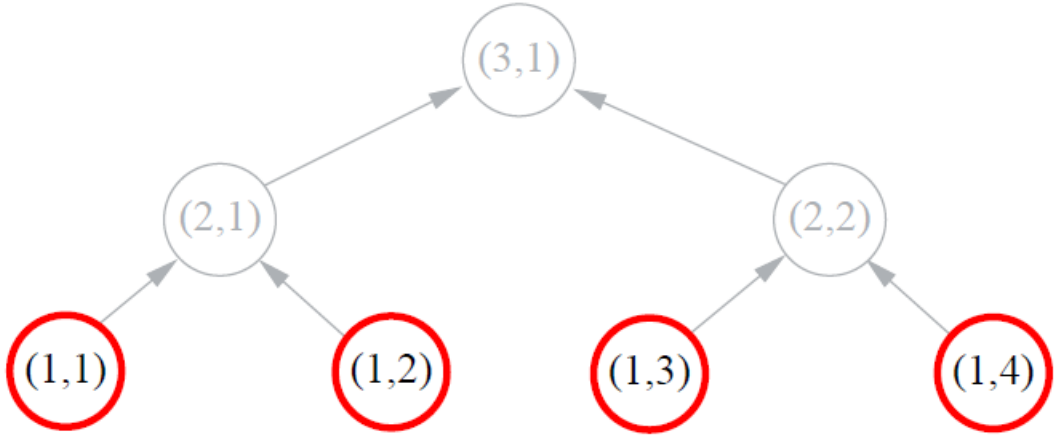
$$\text{After reordering } A \to P^T A P = \left[\begin{array}{cccc|cc|c} A_{11} & 0 & 0 & 0 & A_{15} & 0 & A_{17} \\ 0 & A_{22} & 0 & 0 & A_{25} & 0 & A_{27} \\ 0 & 0 & A_{33} & 0 & 0 & A_{36} & A_{37} \\ 0 & 0 & 0 & A_{44} & 0 & A_{46} & A_{47} \\ \hline A_{51} & A_{52} & 0 & 0 & A_{55} & 0 & A_{57} \\ 0 & 0 & A_{63} & A_{64} & 0 & A_{66} & A_{67} \\ \hline A_{71} & A_{72} & A_{73} & A_{74} & A_{75} & A_{76} & A_{77} \end{array}\right] \tag{1}$$

We can disassemble (1) into following independent matrix structures:

$$A_1 = \left[\begin{array}{c|cc} A_{11} & A_{15} & A_{17} \\ \hline A_{51} & A^1_{55} & A^1_{57} \\ A_{71} & A^1_{75} & A^1_{77} \end{array}\right], \quad A_2 = \left[\begin{array}{c|cc} A_{22} & A_{25} & A_{27} \\ \hline A_{52} & A^2_{55} & A^2_{57} \\ A_{72} & A^2_{75} & A^2_{77} \end{array}\right]$$
$$A_3 = \left[\begin{array}{c|cc} A_{33} & A_{36} & A_{37} \\ \hline A_{63} & A^3_{66} & A^3_{67} \\ A_{73} & A^3_{76} & A^3_{77} \end{array}\right], \quad A_4 = \left[\begin{array}{c|cc} A_{44} & A_{46} & A_{47} \\ \hline A_{64} & A^4_{66} & A^4_{67} \\ A_{74} & A^4_{76} & A^4_{77} \end{array}\right] \tag{2}$$

- These independent sub-matrices can now be processed in parallel.
- Here, we choose $A^1_{55} = A^2_{55} = \frac{1}{2}A_{55}$, $A^1_{57} = A^2_{57} = \frac{1}{2}A_{57}$, $A^3_{66} = A^4_{66} = \frac{1}{2}A_{66}$, $A^3_{67} = A^4_{67} = \frac{1}{2}A_{67}$, $A^1_{77} = A^2_{77} = A^3_{77} = A^4_{77} = \frac{1}{4}A_{77}$.

Parallel Crout ILU factorization at level 1



First-level tasks compute in parallel ILUPACK partial ILUs

$$\left(\begin{array}{c|cc} A_{11}^{(1,i)} & A_{12}^{(1,i)} & A_{13}^{(1,i)} \\ \hline A_{21}^{(1,i)} & A_{22}^{(1,i)} & A_{23}^{(1,i)} \\ A_{31}^{(1,i)} & A_{32}^{(1,i)} & A_{33}^{(1,i)} \end{array}\right) \approx \left(\begin{array}{c|cc} L_{11}^{(1,i)} & 0 & 0 \\ \hline L_{21}^{(1,i)} & I & 0 \\ L_{31}^{(1,i)} & 0 & I \end{array}\right) \left(\begin{array}{c|cc} U_{11}^{(1,i)} & U_{12}^{(1,i)} & U_{13}^{(1,i)} \\ \hline 0 & S_{22}^{(1,i)} & S_{23}^{(1,i)} \\ 0 & S_{32}^{(1,i)} & S_{33}^{(1,i)} \end{array}\right)$$

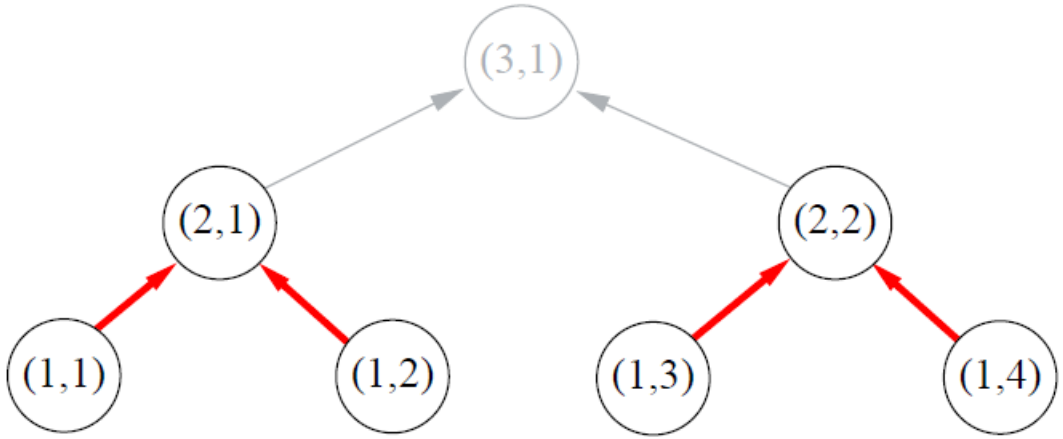
where $i = 1, \ldots, 4$

Abbildung: Parallel Crout ILU factorization [5]

Courtesy: A.F.Martin, Scheduling Strategies for Parallel Sparse Substitution

Fraunhofer ITWM

Parallel Assembly of Schur Complement Matrices at level 2



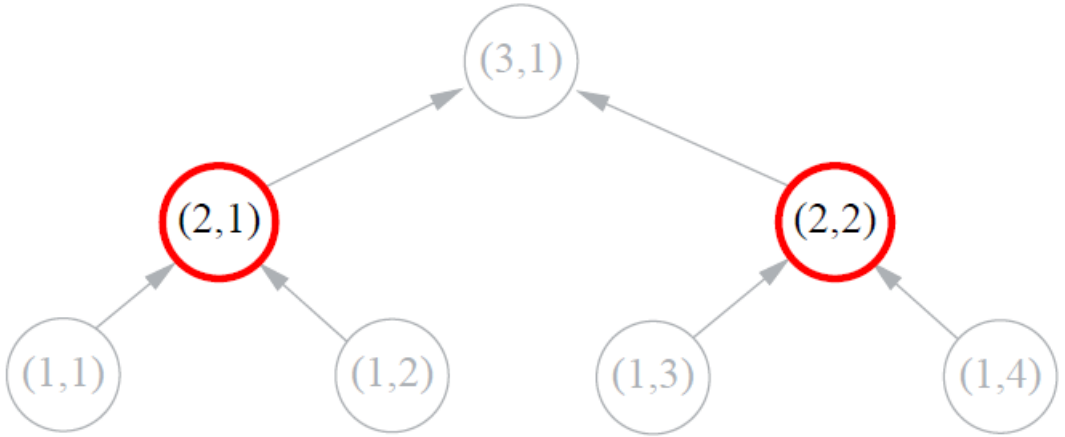
Second-level tasks “merge” the Schur complements of its children

$$\begin{pmatrix} A_{22}^{(2,i)} & A_{23}^{(2,i)} \\ A_{32}^{(2,i)} & A_{33}^{(2,i)} \end{pmatrix} = \begin{pmatrix} S_{22}^{(1,2i-1)} & S_{23}^{(1,2i-1)} \\ S_{32}^{(1,2i-1)} & S_{33}^{(1,2i-1)} \end{pmatrix} + \begin{pmatrix} S_{22}^{(1,2i)} & S_{23}^{(1,2i)} \\ S_{32}^{(1,2i)} & S_{33}^{(1,2i)} \end{pmatrix},$$

where $i = 1, 2$

Fraunhofer ITWM

Parallel Crout ILU factorization at level 2



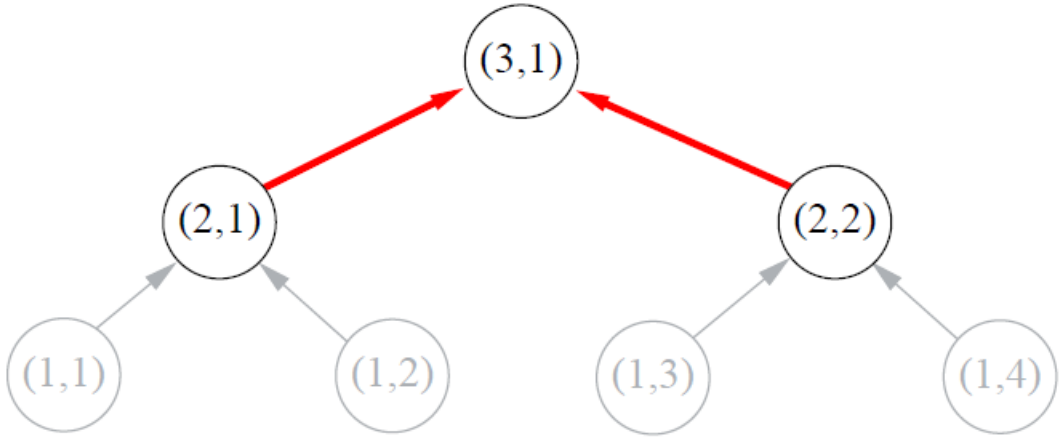
Second-level tasks compute in parallel ILUPACK partial ILUs,

$$\left(\begin{array}{c|c} A_{22}^{(2,i)} & A_{23}^{(2,i)} \\ \hline A_{32}^{(2,i)} & A_{33}^{(2,i)} \end{array}\right) \approx \left(\begin{array}{c|c} L_{22}^{(2,i)} & 0 \\ \hline L_{32}^{(2,i)} & I \end{array}\right) \left(\begin{array}{c|c} U_{22}^{(2,i)} & U_{23}^{(2,i)} \\ \hline 0 & S_{33}^{(2,i)} \end{array}\right),$$

where now $i = 1, 2$

Fraunhofer ITWM

Assembly of Schur Complement Matrices at level 3

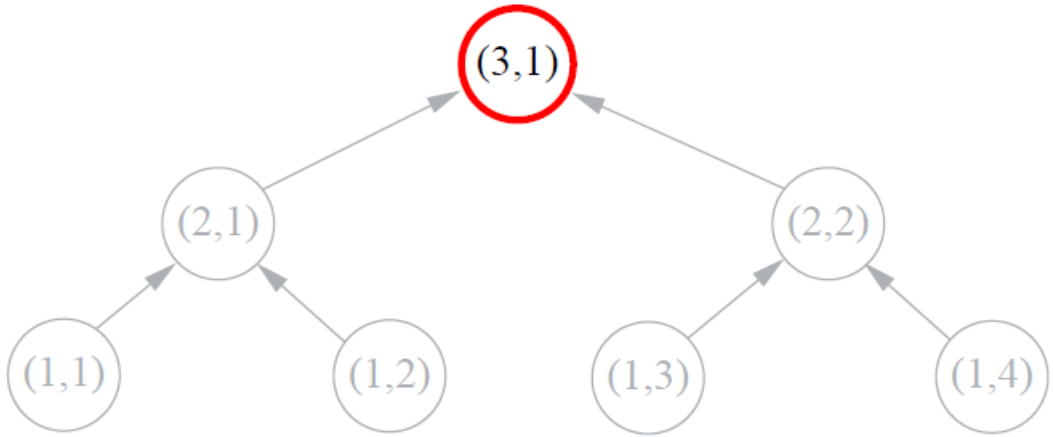


The root task "merges" the Schur complements of its children

$$A_{33}^{(3,1)} = S_{33}^{(2,1)} + S_{33}^{(2,2)}$$

Fraunhofer ITWM

Crout complete ILU factorization at level 3



Finally, the root task completes the parallel ILU

$$A_{33}^{(3,1)} \approx L_{33}^{(3,1)} U_{33}^{(3,1)}$$

Fraunhofer ITWM

Why Sparse Vector Update matters to us?

- Serial Crout ILU factorization takes place at every node in the task tree. Therefore its efficient implementation is crucial for the parallel factorization.
- Sparse vector z and w update is most time consuming kernel in serial factorization. Hence its implementation should be properly investigated.

Algorithm 4 Crout version of ILU Algorithm: ILUC

1: **for** $k = 1 : n$ **do**
2: $z_{1:k-1} = 0, \quad z_{k:n} = a_{k,k:n}$ ▷ Initialize row z
3: **for** $(1 \leq i \leq k-1$ **and** $l_{ki} \neq 0)$ **do**
4: $z_{k:n} = z_{k:n} - l_{ki} * u_{i,k:n}$ ▷ Preparation to update $u_{k,:}$
5: **end do**
6: $w_{1:k} = 0, \quad w_{k+1:n} = a_{k+1:n,k}$ ▷ Initialize column z
7: **for** $(1 \leq i \leq k-1$ **and** $u_{ik} \neq 0)$ **do**
8: $w_{k+1:n} = w_{k+1:n} - u_{ik} * l_{k+1:n,i}$ ▷ Preparation to update $l_{:,k}$
9: **end do**
10: Apply a dropping rule to row z
11: Apply a dropping rule to column w
12: $u_{k,:} = z$
13: $l_{:,k} = w/u_{kk}, \quad l_{kk} = 1$
14: **end do**

Handling Sparsity at the Step k

- Row vector z is updated using the linear combination of preceding rows of U.

Algorithm 5 Update in sparse vector z

1: **for** $(1 \leq i \leq k-1$ **and** $l_{ki} \neq 0)$ **do**
2: $\quad z_{k:n} = z_{k:n} - l_{ki} * u_{i,k:n}$
3: **end do**

- We only store the non-zero entries in the sparse vector z.
- Features required in sparse Vector z:
  - Column Index j lookup: Should be O(1) for faster lookups,
  - Insertion of entry $z[j]$: Should be O(1),
  - Erasing entry $z[j]$: Should be O(1),
  - Sorting z based on the column index j: Avg case O(nlog(n)).

Handling Sparsity at the Step k

- Row vector z is updated using the linear combination of preceding rows of U.

Algorithm 6 Update in sparse vector z

1: **for** $(1 \leq i \leq k-1$ **and** $l_{ki} \neq 0)$ **do**
2: $\quad z_{k:n} = z_{k:n} - l_{ki} * u_{i,k:n}$
3: **end do**

- We only store the non-zero entries in the sparse vector z.
- Features required in sparse Vector z:
  - Column Index j lookup: Should be O(1) for faster lookups,
  - Insertion of entry $z[j]$: Should be O(1),
  - Erasing entry $z[j]$: Should be O(1),
  - Sorting z based on the column index j: Avg case O(nlog(n)).

Handling Sparsity at the Step k

- Row vector z is updated using the linear combination of preceding rows of U.

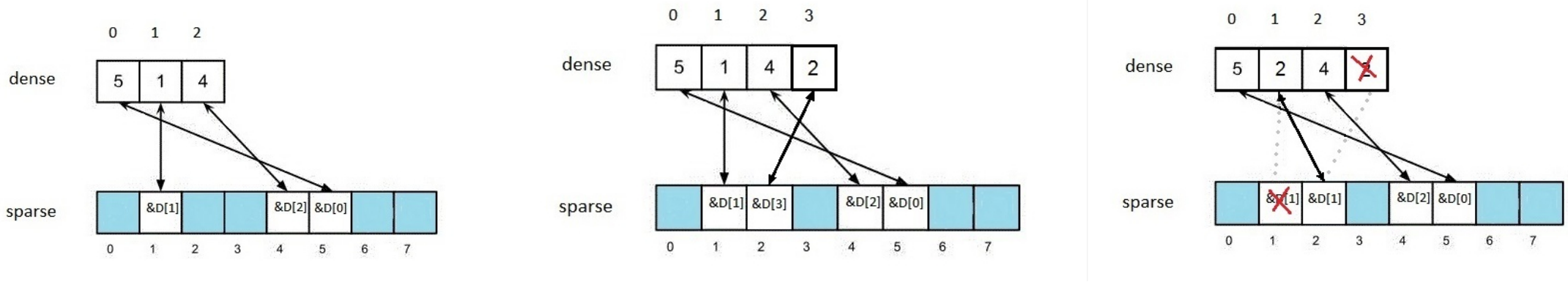
Algorithm 7 Update in sparse vector z

1: **for** $(1 \leq i \leq k-1$ **and** $l_{ki} \neq 0)$ **do**
2: $\quad z_{k:n} = z_{k:n} - l_{ki} * u_{i,k:n}$
3: **end do**

- We only store the non-zero entries in the sparse vector z.
- Features required in sparse Vector z:
  - Column Index j lookup: Should be O(1) for faster lookups,
  - Insertion of entry $z[j]$: Should be O(1),
  - Erasing entry $z[j]$: Should be O(1),
  - Sorting z based on the column index j: Avg case O(nlog(n)).

Different data structures that implements the sparse vector z

- We have used std::map, std::unordered_map and a custom implementation to implement sparse vector z.
- Custom Implementation:



(a) Dense (length n) and sparse vector (length N)

(b) Insertion at index 2

(c) Deletion at index 1

[2] Fast Implementations of Sparse Sets in C++
https://www.codeproject.com/Articles/859324/Fast-Implementations-of-Sparse-Sets-in-Cplusplus

Complexity of different data structures

	std::map	std::unordered_map	Custom Implementation
Lookup (find)	O(log(n))	O(n) worst case, O(1) avg case	O(1)
Insertion	O(log(n))	O(n) worst case, O(1) avg case	O(1)
Deletion (erase)	O(log(n))	O(n) worst case, O(1) avg case	O(1)
Sort	Already Sorted	O(nlogn)	O(nlogn)

Tabelle: Time complexity

	std::map	std::unordered_map	Custom Implementation
Space Complexity	O(n)	O(n)	O(N) + O(n)

Tabelle: Space complexity

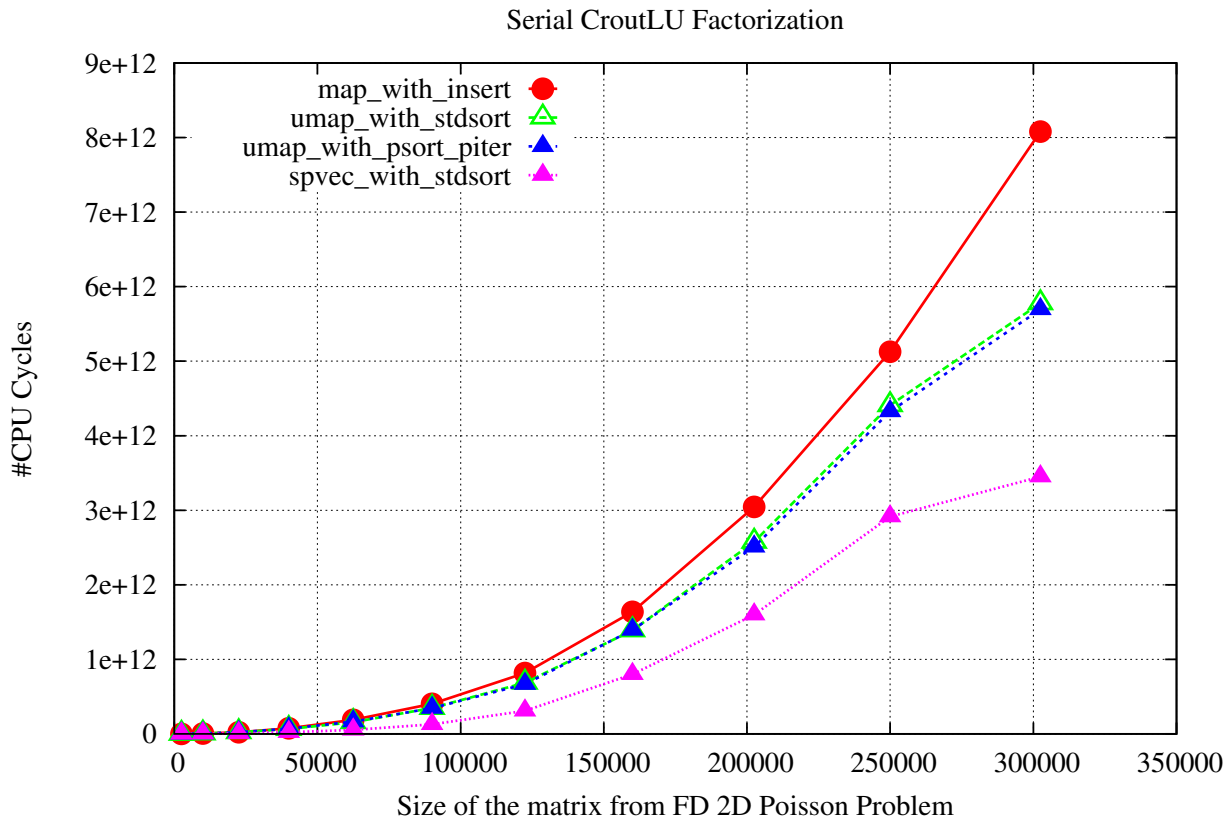
- n: length of the dense vector,
- N: length of the sparse vector ($n << N$).

Computational cost to generate the sparse vector z

	std::map	**std::unordered_map**	**Custom Implementation**
Fill sparse vector z with A(k,k:n)	O(C_0)	O(C_0)	O(C_0)
Lookup and insert	nlog(n) + O(n)	O(n^2) in worst case, O(n) in avg case	O(n)
Dropping (tau + p)	2*O(n) + n*log(p)	2*O(n) + n*log(p)	2*O(n) + n*log(p)
Sort based on index	Already sorted	O(plog(p))	O(plog(p))
Total Computation Cost	**nlog(n) + log(p)*n + O(n)**	**Worst Case: O(n^2) + log(p)*n + O(n) Avg Case: O(log(p)*n) + O(n)**	**log(p)*n + O(n)**

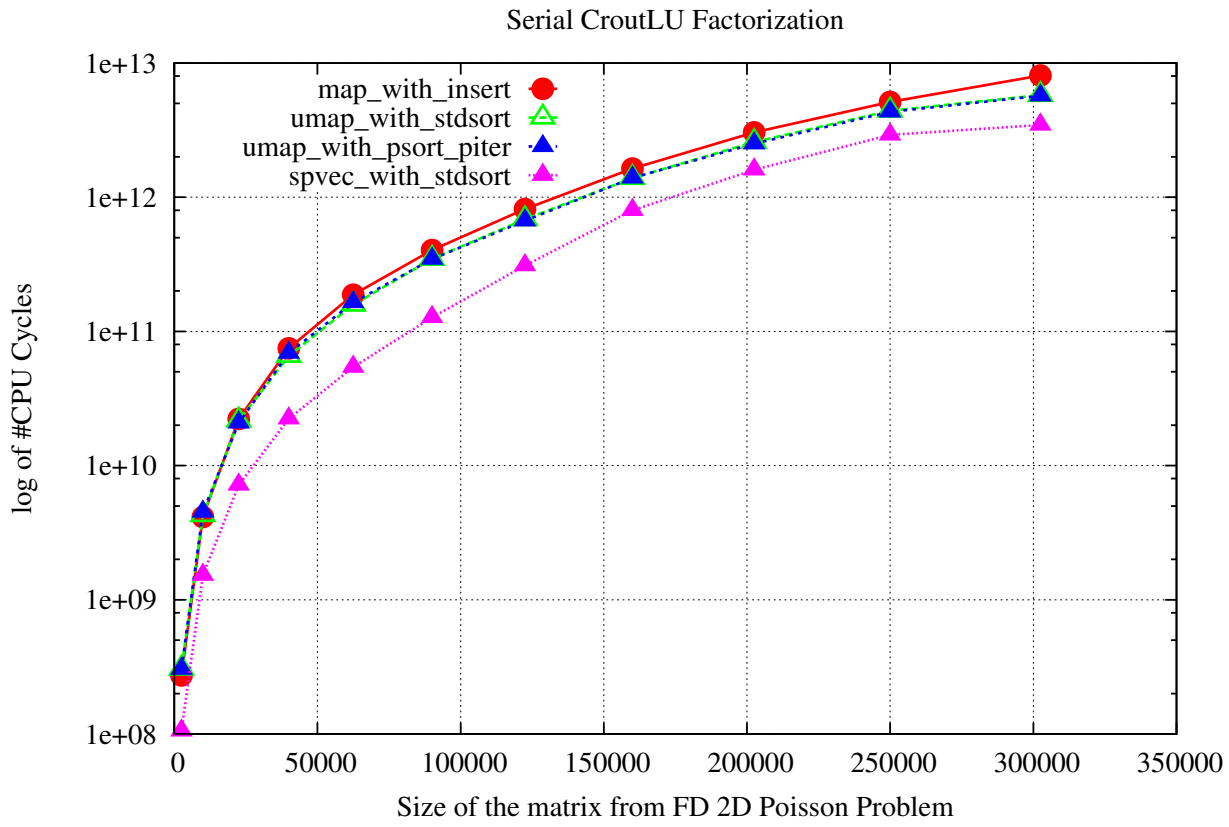
- p = constant in general, $p = n$ in case of full fill-in in z.

Computational Time vs Matrix Size



Fraunhofer ITWM

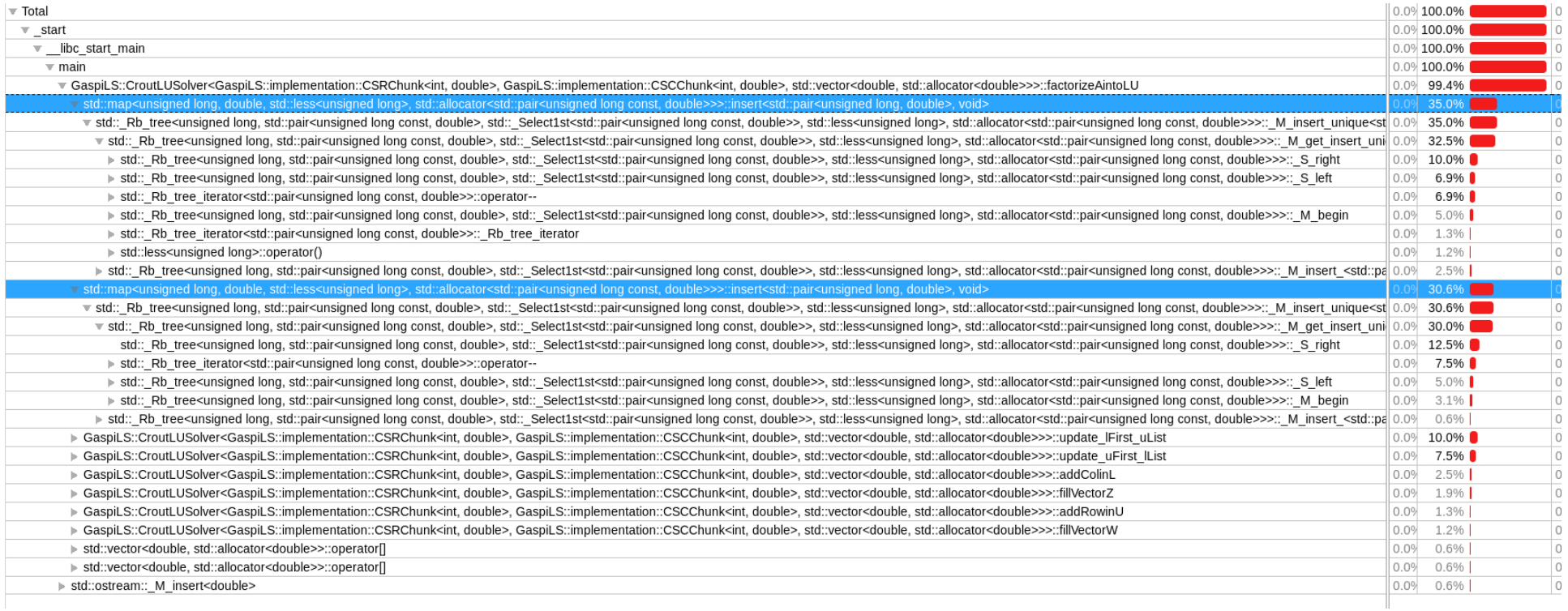
Log of Computational Time vs Matrix Size



Fraunhofer ITWM

Profiler reading: using std::map to store the sparse vector

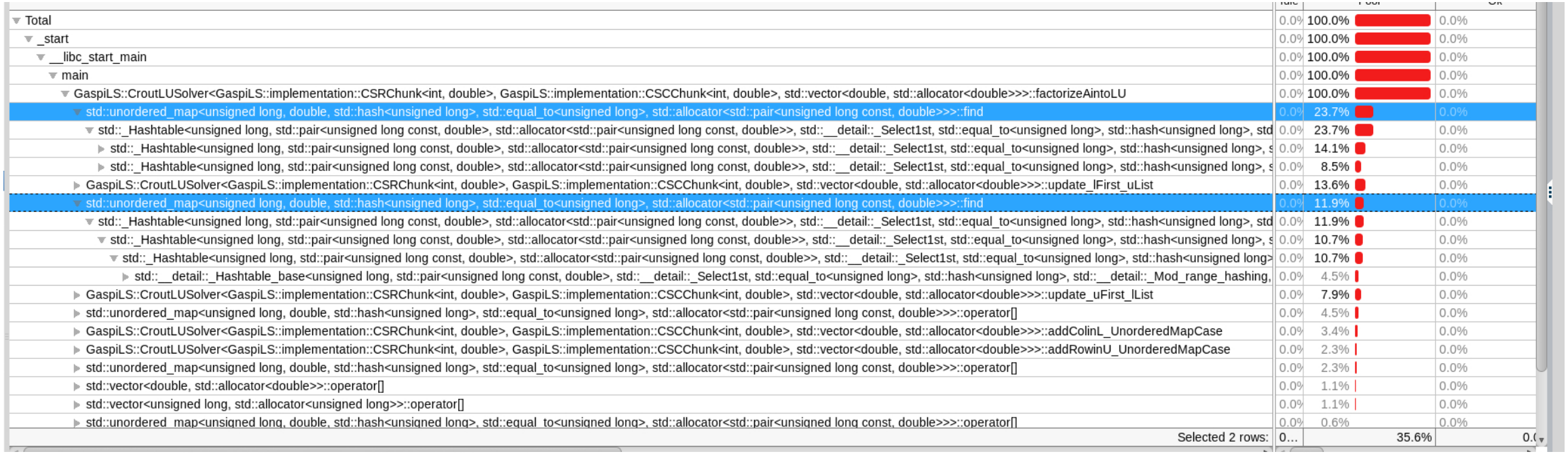
insert in std::map takes up **65.6 %** time of the Crout LU factorization.



Fraunhofer ITWM

Profiler reading: using std::unordered_map to store the sparse vector

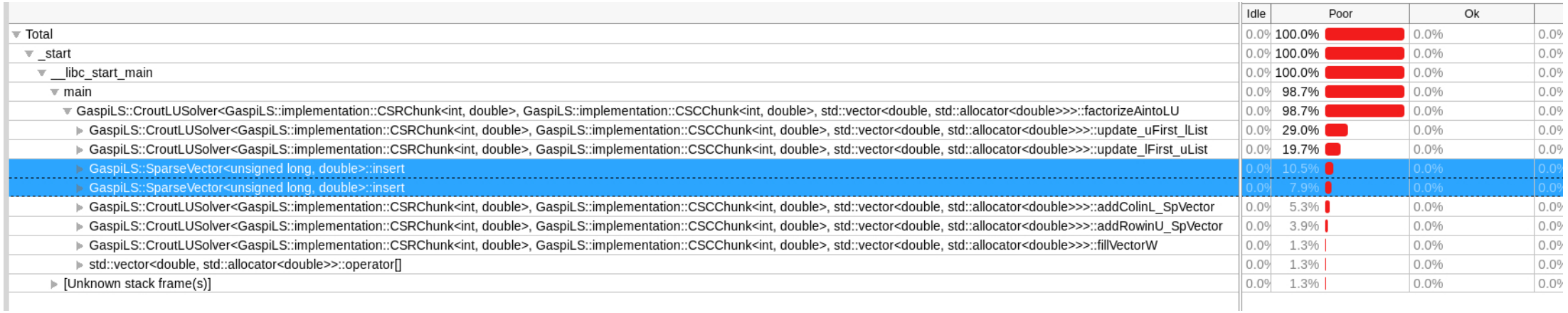
find in std::unorderd_map takes up **35.6 %** time of the Crout LU factorization.



Total	0.0%	100.0%	0.0%
_start	0.0%	100.0%	0.0%
__libc_start_main	0.0%	100.0%	0.0%
main	0.0%	100.0%	0.0%
GaspiLS::CroutLUSolver<GaspiLS::implementation::CSRChunk<int, double>, GaspiLS::implementation::CSCChunk<int, double>, std::vector<double, std::allocator<double>>>::factorizeAintoLU	0.0%	100.0%	0.0%
std::unordered_map<unsigned long, double, std::hash<unsigned long>, std::equal_to<unsigned long>, std::allocator<std::pair<unsigned long const, double>>>::find	0.0%	23.7%	0.0%
std::_Hashtable<unsigned long, std::pair<unsigned long const, double>, std::allocator<std::pair<unsigned long const, double>>, std::__detail::_Select1st, std::equal_to<unsigned long>, std::hash<unsigned long>, std	0.0%	23.7%	0.0%
std::_Hashtable<unsigned long, std::pair<unsigned long const, double>, std::allocator<std::pair<unsigned long const, double>>, std::__detail::_Select1st, std::equal_to<unsigned long>, std::hash<unsigned long>, s	0.0%	14.1%	0.0%
std::_Hashtable<unsigned long, std::pair<unsigned long const, double>, std::allocator<std::pair<unsigned long const, double>>, std::__detail::_Select1st, std::equal_to<unsigned long>, std::hash<unsigned long>, s	0.0%	8.5%	0.0%
GaspiLS::CroutLUSolver<GaspiLS::implementation::CSRChunk<int, double>, GaspiLS::implementation::CSCChunk<int, double>, std::vector<double, std::allocator<double>>>::update_lFirst_uList	0.0%	13.6%	0.0%
std::unordered_map<unsigned long, double, std::hash<unsigned long>, std::equal_to<unsigned long>, std::allocator<std::pair<unsigned long const, double>>>::find	0.0%	11.9%	0.0%
std::_Hashtable<unsigned long, std::pair<unsigned long const, double>, std::allocator<std::pair<unsigned long const, double>>, std::__detail::_Select1st, std::equal_to<unsigned long>, std::hash<unsigned long>, std	0.0%	11.9%	0.0%
std::_Hashtable<unsigned long, std::pair<unsigned long const, double>, std::allocator<std::pair<unsigned long const, double>>, std::__detail::_Select1st, std::equal_to<unsigned long>, std::hash<unsigned long>, s	0.0%	10.7%	0.0%
std::_Hashtable<unsigned long, std::pair<unsigned long const, double>, std::allocator<std::pair<unsigned long const, double>>, std::__detail::_Select1st, std::equal_to<unsigned long>, std::hash<unsigned long>	0.0%	10.7%	0.0%
std::__detail::_Hashtable_base<unsigned long, std::pair<unsigned long const, double>, std::__detail::_Select1st, std::equal_to<unsigned long>, std::hash<unsigned long>, std::__detail::_Mod_range_hashing,	0.0%	4.5%	0.0%
GaspiLS::CroutLUSolver<GaspiLS::implementation::CSRChunk<int, double>, GaspiLS::implementation::CSCChunk<int, double>, std::vector<double, std::allocator<double>>>::update_uFirst_lList	0.0%	7.9%	0.0%
std::unordered_map<unsigned long, double, std::hash<unsigned long>, std::equal_to<unsigned long>, std::allocator<std::pair<unsigned long const, double>>>::operator[]	0.0%	4.5%	0.0%
GaspiLS::CroutLUSolver<GaspiLS::implementation::CSRChunk<int, double>, GaspiLS::implementation::CSCChunk<int, double>, std::vector<double, std::allocator<double>>>::addColinL_UnorderedMapCase	0.0%	3.4%	0.0%
GaspiLS::CroutLUSolver<GaspiLS::implementation::CSRChunk<int, double>, GaspiLS::implementation::CSCChunk<int, double>, std::vector<double, std::allocator<double>>>::addRowinU_UnorderedMapCase	0.0%	2.3%	0.0%
std::unordered_map<unsigned long, double, std::hash<unsigned long>, std::equal_to<unsigned long>, std::allocator<std::pair<unsigned long const, double>>>::operator[]	0.0%	2.3%	0.0%
std::vector<double, std::allocator<double>>::operator[]	0.0%	1.1%	0.0%
std::vector<unsigned long, std::allocator<unsigned long>>::operator[]	0.0%	1.1%	0.0%
std::unordered_map<unsigned long, double, std::hash<unsigned long>, std::equal_to<unsigned long>, std::allocator<std::pair<unsigned long const, double>>>::operator[]	0.0%	0.6%	0.0%
Selected 2 rows:	0...	35.6%	0.(

Profiler reading: Custom implementation for the sparse vector

insert in the custom implementation takes up only **18.4 %** time of the Crout LU factorization.



	Idle	Poor	Ok	
Total	0.0%	100.0%	0.0%	0.0%
_start	0.0%	100.0%	0.0%	0.0%
__libc_start_main	0.0%	100.0%	0.0%	0.0%
main	0.0%	98.7%	0.0%	0.0%
GaspiLS::CroutLUSolver<GaspiLS::implementation::CSRChunk<int, double>, GaspiLS::implementation::CSCChunk<int, double>, std::vector<double, std::allocator<double>>>::factorizeAintoLU	0.0%	98.7%	0.0%	0.0%
GaspiLS::CroutLUSolver<GaspiLS::implementation::CSRChunk<int, double>, GaspiLS::implementation::CSCChunk<int, double>, std::vector<double, std::allocator<double>>>::update_uFirst_lList	0.0%	29.0%	0.0%	0.0%
GaspiLS::CroutLUSolver<GaspiLS::implementation::CSRChunk<int, double>, GaspiLS::implementation::CSCChunk<int, double>, std::vector<double, std::allocator<double>>>::update_lFirst_uList	0.0%	19.7%	0.0%	0.0%
GaspiLS::SparseVector<unsigned long, double>::insert	0.0%	10.5%	0.0%	0.0%
GaspiLS::SparseVector<unsigned long, double>::insert	0.0%	7.9%	0.0%	0.0%
GaspiLS::CroutLUSolver<GaspiLS::implementation::CSRChunk<int, double>, GaspiLS::implementation::CSCChunk<int, double>, std::vector<double, std::allocator<double>>>::addColinL_SpVector	0.0%	5.3%	0.0%	0.0%
GaspiLS::CroutLUSolver<GaspiLS::implementation::CSRChunk<int, double>, GaspiLS::implementation::CSCChunk<int, double>, std::vector<double, std::allocator<double>>>::addRowinU_SpVector	0.0%	3.9%	0.0%	0.0%
GaspiLS::CroutLUSolver<GaspiLS::implementation::CSRChunk<int, double>, GaspiLS::implementation::CSCChunk<int, double>, std::vector<double, std::allocator<double>>>::fillVectorW	0.0%	1.3%	0.0%	0.0%
std::vector<double, std::allocator<double>>::operator[]	0.0%	1.3%	0.0%	0.0%
[Unknown stack frame(s)]	0.0%	1.3%	0.0%	0.0%

Summary and future outlook

- Understood Multilevel Parallel Crout ILU factorization algorithm of matrix A.
- Insert in Sparse vector is the bottleneck in Serial implementation.
- Optimized using custom implementation of Sparse vector.
- Since the serial implementation looks good (reasonably optimal), we will start parallel implentation.

Questions/Discussion?

Fraunhofer ITWM

References

- `http://www.gaspils.de/.`
- `https://www.codeproject.com/Articles/859324/Fast-Implementations-of-Sparse-Sets-in-Cplusplus.`
- José Ignacio Aliaga, Rosa M. Badia, Maria Barreda, Matthias Bollhöfer, Ernesto Dufrechou, Pablo Ezzatti, and Enrique S. Quintana-Ortí.
  Exploiting task and data parallelism in ilupack's preconditioned CG solver on NUMA architectures and many-core accelerators.
  Parallel Computing, 54:97–107, 2016.
- Daniel Gruenewald Christian Simmendinger Mirko Rahn.
  Gaspi tutorial.
- Alberto F. Martin Enrique S. Quintana-Orti Jose I. Aliaga, Matthias Bollheofer.
  Scheduling strategies for parallel sparse backward/forward substitution.