Data-flow Parallel Implementation with OmpSs of ILUPACK

J.I. Aliaga[1] R.M. Badía[2] M. Barreda[1] M. Bollhöfer[3]
E.S. Quintana-Ortí[1]

[1]Deparment of Computer Science and Engineering, Univ. Jaume I (Spain)
{aliaga,mvaya,quintana}@icc.uji.es

[2]Barcelona Supercomputing Center (BSC-CNS) (Spain).
rosa.m.badia@bsc.es

[3]Institute of Computational Mathematics, TU-Braunschweig (Germany)
m.bollhoefer@tu-bs.de

July, 2014

Motivation and Introduction

- We consider the **efficient** solution of linear systems

$$Ax = b$$

  where
  - $A \in \mathbb{R}^{n,n}$ is a **large** and **sparse** SPD coefficient matrix
  - $x \in \mathbb{R}^n$ is the sought-after solution
  - $b \in \mathbb{R}^n$ is a given r.h.s. vector
- The solution of this problem
  - arises in very many large-scale application problems
  - is frequently the most time-consuming part of the computation

Motivation and Introduction

Two main approaches to “attack” the problem:

- Direct methods (sparse Cholesky, sparse LU, etc.) may fail due to excessive fill-in of the factors
- Iterative methods (CG, GMRES, etc.) may present slow convergence or even divergence

Convergence of iterative methods can be accelerated via preconditioning:

- Incomplete LU factorization
- Sparse approximate inverse
- ...

Motivation and Introduction

Two main approaches to “attack” the problem:

- Direct methods (sparse Cholesky, sparse LU, etc.) may fail due to excessive fill-in of the factors
- Iterative methods (CG, GMRES, etc.) may present slow convergence or even divergence

Convergence of iterative methods can be accelerated via preconditioning:

- Incomplete LU factorization
- Sparse approximate inverse
- . . .

Motivation and Introduction

We propose to solve $Ax = b$ **iteratively** using

- Preconditioned Conjugate Gradient (PCG)
  - Among the best iterative approaches to solve SPD linear systems
- ILUPACK preconditioning techniques
  - Incomplete Cholesky (IC) factorization-based preconditioners
    $\rightarrow$ Multilevel IC (MIC) preconditioners: **Inverse-based** approach
  - Successful for a wide range of large-scale application problems of scientific interest with up to million of equations

Motivation and Introduction

We propose to solve $Ax = b$ **iteratively** using

- Preconditioned Conjugate Gradient (PCG)
  - Among the best iterative approaches to solve SPD linear systems
- ILUPACK preconditioning techniques
  - Incomplete Cholesky (IC) factorization-based preconditioners
    $\rightarrow$ Multilevel IC (MIC) preconditioners: **Inverse-based** approach
  - Successful for a wide range of large-scale application problems of scientific interest with up to million of equations

Motivation and Introduction

- The large scale of the problems faced by ILUPACK makes it evident the need to reduce the time-to-solution via parallel computing techniques
- Development of parallel ILUPACK-based solvers
  - Computation of adjacency graph G_A
  - Parallelism revealed by nested dissection on G_A
    $\rightarrow$ recursive application of vertex separator partitioning
  - Dynamic scheduling to improve load-balancing during execution
    $\rightarrow$ run-time tuned for a specific platform/application problem
  - OmpSs is a data-flow programming model
    $\rightarrow$ Exploit the data-dependencies between the tasks
    $\rightarrow$ Useful to combine different platforms in heterogenous systems

Motivation and Introduction

- The large scale of the problems faced by ILUPACK makes it evident the need to reduce the time-to-solution via parallel computing techniques
- Development of parallel ILUPACK-based solvers
  - Computation of adjacency graph G_A
  - Parallelism revealed by nested dissection on G_A
    $\rightarrow$ recursive application of vertex separator partitioning
  - Dynamic scheduling to improve load-balancing during execution
    $\rightarrow$ run-time tuned for a specific platform/application problem
  - OmpSs is a data-flow programming model
    $\rightarrow$ Exploit the data-dependencies between the tasks
    $\rightarrow$ Useful to combine different platforms in heterogenous systems

Motivation and Introduction

- The large scale of the problems faced by ILUPACK makes it evident the need to reduce the time-to-solution via parallel computing techniques
- Development of parallel ILUPACK-based solvers
  - Computation of adjacency graph G_A
  - Parallelism revealed by nested dissection on G_A
    $\rightarrow$ recursive application of vertex separator partitioning
  - Dynamic scheduling to improve load-balancing during execution
    $\rightarrow$ run-time tuned for a specific platform/application problem
  - OmpSs is a data-flow programming model
    $\rightarrow$ Exploit the data-dependencies between the tasks
    $\rightarrow$ Useful to combine different platforms in heterogenous systems

Outline

Outline

Inverse-based IC preconditioners

- The (root-free) sparse Cholesky decomp. computes $L \in \mathbb{R}^{n,n}$, sparse unit lower triangular, and $D \in \mathbb{R}^{n,n}$, diagonal with positive entries s.t.

$$A = LDL^T$$

- In the Incomplete Cholesky (IC) factorization A is factored as

$$A = \tilde{L}\tilde{D}\tilde{L}^T + E$$

  where $E \in \mathbb{R}^{n,n}$ is a "small" perturbation matrix consisting of those entries dropped during factorization
- $M = \tilde{L}\tilde{D}\tilde{L}^T$ is applied to the original system $Ax = b$ in order to accelerate the convergence of the PCG solver

$$(\tilde{D}^{-1/2}\tilde{L}^{-1})A(\tilde{L}^{-T}\tilde{D}^{-1/2})$$

- Inverse-based IC factorizations construct $M = \tilde{L}\tilde{D}\tilde{L}^T$ s.t. $\|\tilde{L}^{-1}\| \leq \kappa$
  - $\kappa = 5$ or $\kappa = 10$ are good choices for many problems
  - Pivoting and **multilevel** methods are required in general

Inverse-based IC preconditioners

- The (root-free) sparse Cholesky decomp. computes $L \in \mathbb{R}^{n,n}$, sparse unit lower triangular, and $D \in \mathbb{R}^{n,n}$, diagonal with positive entries s.t.

$$A = LDL^T$$

- In the Incomplete Cholesky (IC) factorization A is factored as

$$A = \tilde{L}\tilde{D}\tilde{L}^T + E$$

  where $E \in \mathbb{R}^{n,n}$ is a "small" perturbation matrix consisting of those entries dropped during factorization

- $M = \tilde{L}\tilde{D}\tilde{L}^T$ is applied to the original system $Ax = b$ in order to accelerate the convergence of the PCG solver

$$(\tilde{D}^{-1/2}\tilde{L}^{-1})A(\tilde{L}^{-T}\tilde{D}^{-1/2})$$

- Inverse-based IC factorizations construct $M = \tilde{L}\tilde{D}\tilde{L}^T$ s.t. $\|\tilde{L}^{-1}\| \leq \kappa$
  - $\kappa = 5$ or $\kappa = 10$ are good choices for many problems
  - Pivoting and **multilevel** methods are required in general

Inverse-based IC preconditioners

- The (root-free) sparse Cholesky decomp. computes $L \in \mathbb{R}^{n,n}$, sparse unit lower triangular, and $D \in \mathbb{R}^{n,n}$, diagonal with positive entries s.t.
- In the Incomplete Cholesky (IC) factorization A is factored as

$$A = \tilde{L}\tilde{D}\tilde{L}^T + E$$

  where $E \in \mathbb{R}^{n,n}$ is a "small" perturbation matrix consisting of those entries dropped during factorization
- $M = \tilde{L}\tilde{D}\tilde{L}^T$ is applied to the original system $Ax = b$ in order to accelerate the convergence of the PCG solver

$$(\tilde{D}^{-1/2}\tilde{L}^{-1})A(\tilde{L}^{-T}\tilde{D}^{-1/2})$$

- Inverse-based IC factorizations construct $M = \tilde{L}\tilde{D}\tilde{L}^T$ s.t. $\|\tilde{L}^{-1}\| \leq \kappa$
  - $\kappa = 5$ or $\kappa = 10$ are good choices for many problems
  - Pivoting and **multilevel** methods are required in general

Inverse-based IC preconditioners

- The (root-free) sparse Cholesky decomp. computes $L \in \mathbb{R}^{n,n}$, sparse unit lower triangular, and $D \in \mathbb{R}^{n,n}$, diagonal with positive entries s.t.
- In the Incomplete Cholesky (IC) factorization A is factored as

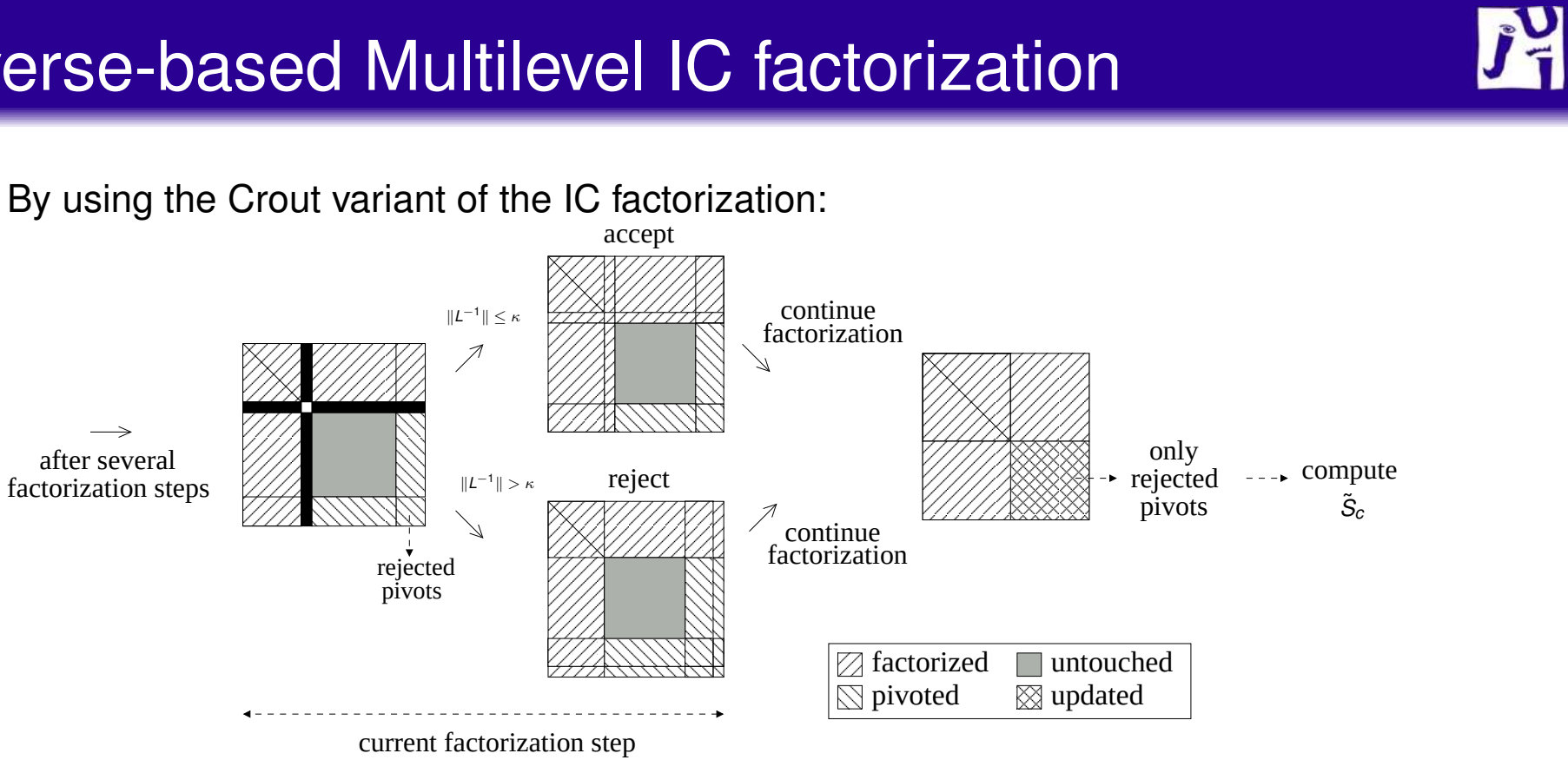
$$A = \tilde{L}\tilde{D}\tilde{L}^T + E$$

  where $E \in \mathbb{R}^{n,n}$ is a "small" perturbation matrix consisting of those entries dropped during factorization
- $M = \tilde{L}\tilde{D}\tilde{L}^T$ is applied to the original system $Ax = b$ in order to accelerate the convergence of the PCG solver

$$(\tilde{D}^{-1/2}\tilde{L}^{-1})A(\tilde{L}^{-T}\tilde{D}^{-1/2})$$

- Inverse-based IC factorizations construct $M = \tilde{L}\tilde{D}\tilde{L}^T$ s.t. $\|\tilde{L}^{-1}\| \leq \kappa$
  - $\kappa = 5$ or $\kappa = 10$ are good choices for many problems
  - Pivoting and **multilevel** methods are required in general

Inverse-based Multilevel IC factorization

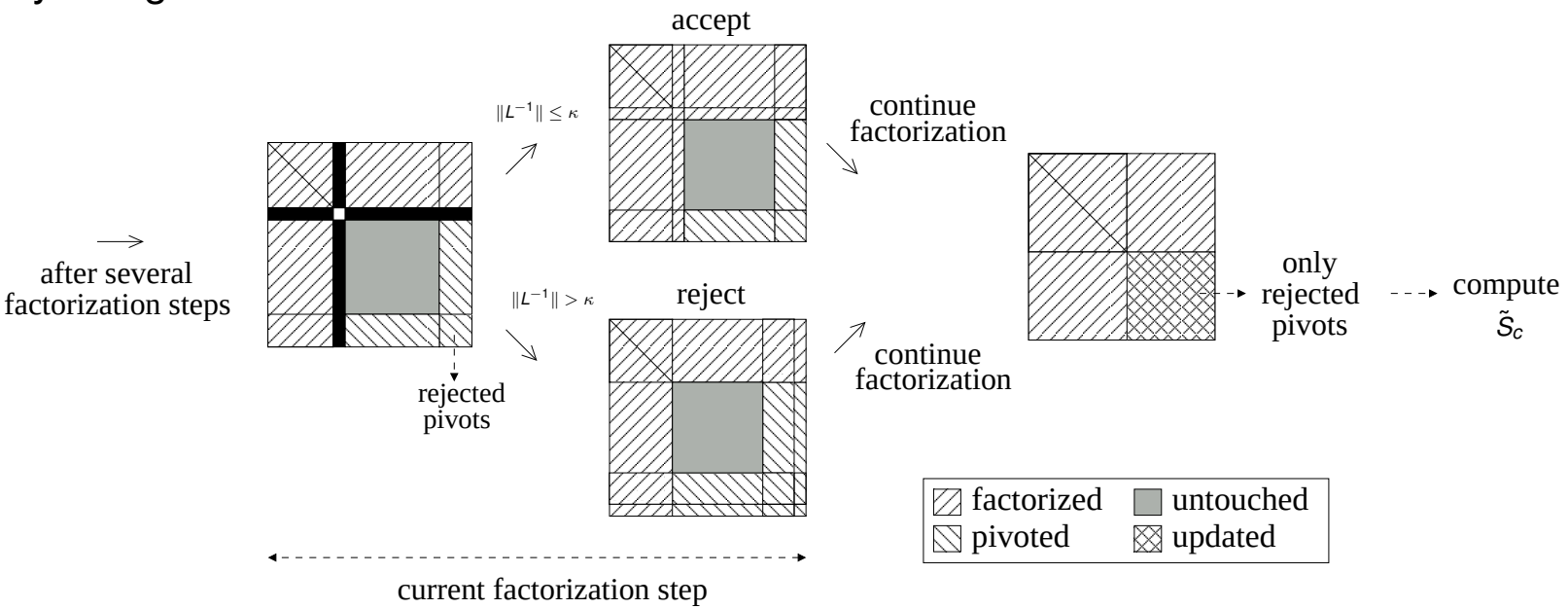


By using the Crout variant of the IC factorization:

- The IC decomp. employs **Inverse-based** pivoting
  → Row/Col. k of $\tilde{L}$ / $\tilde{L}^T$ obtained at step k along with estimation $t_k \approx \|e_k^T \tilde{L}^{-1}\|$
  → Row/Col. s.t. $t_k > \kappa$ are moved to the bottom/right-end of the matrix
- The IC factorization **stops** when the trailing block only contains rejected pivots
  → The computation of the approximate Schur complement $\tilde{S}_c$ completes . . .

Inverse-based Multilevel IC factorization

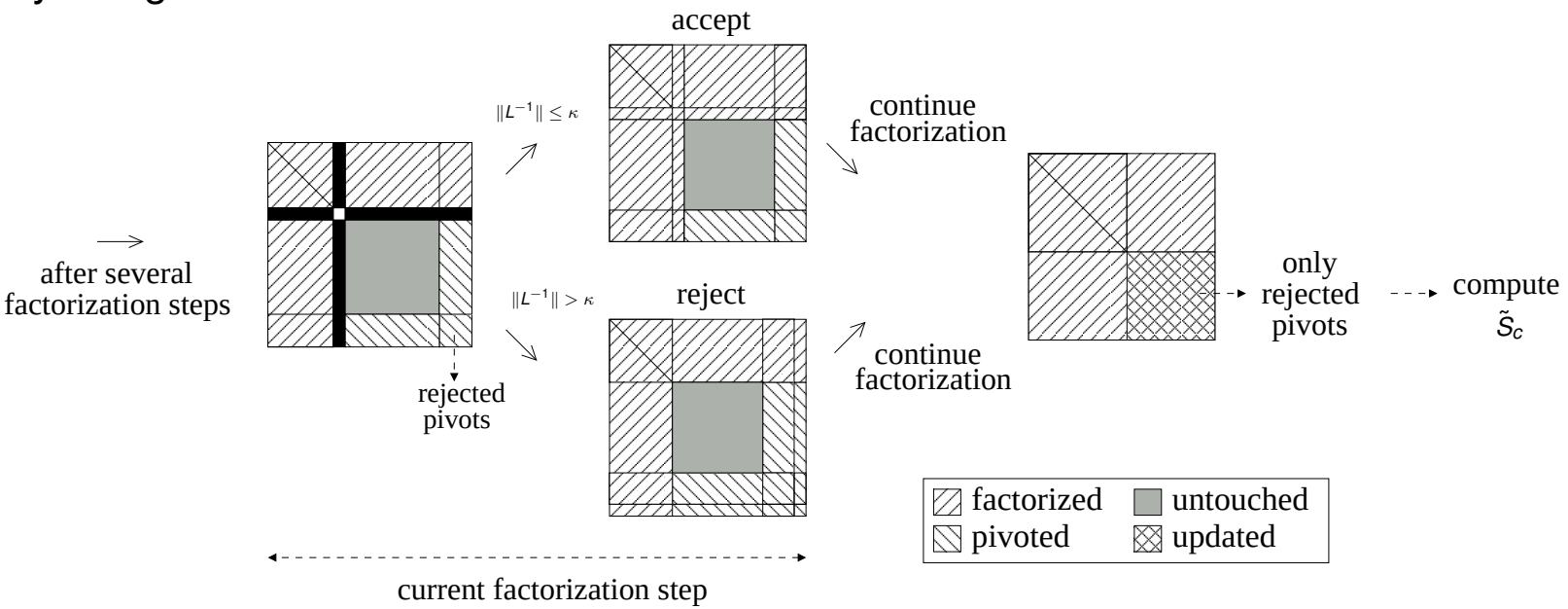
By using the Crout variant of the IC factorization:



- The IC decomp. employs **Inverse-based** pivoting
  → Row/Col. k of $\tilde{L}$ / $\tilde{L}^T$ obtained at step k along with estimation $t_k \approx \|e_k^T \tilde{L}^{-1}\|$
  → Row/Col. s.t. $t_k > \kappa$ are moved to the bottom/right-end of the matrix
- The IC factorization **stops** when the trailing block only contains rejected pivots
  → The computation of the approximate Schur complement $\tilde{S}_c$ completes . . .

Inverse-based Multilevel IC factorization

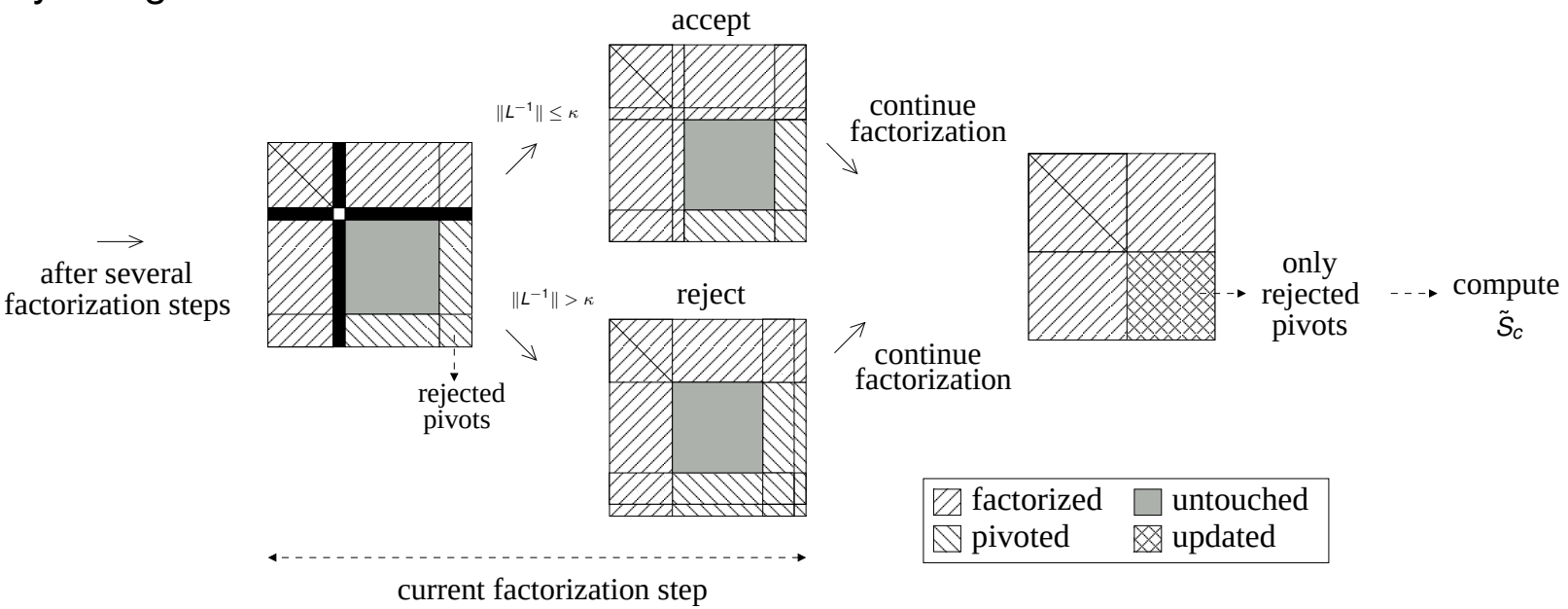
By using the Crout variant of the IC factorization:



- The IC decomp. employs **Inverse-based** pivoting
  → Row/Col. k of $\tilde{L}$ / $\tilde{L}^T$ obtained at step k along with estimation $t_k \approx \|e_k^T \tilde{L}^{-1}\|$
  → Row/Col. s.t. $t_k > \kappa$ are moved to the bottom/right-end of the matrix
- The IC factorization **stops** when the trailing block only contains rejected pivots
  → The computation of the approximate Schur complement $\tilde{S}_c$ completes . . .

Inverse-based Multilevel IC factorization

By using the Crout variant of the IC factorization:



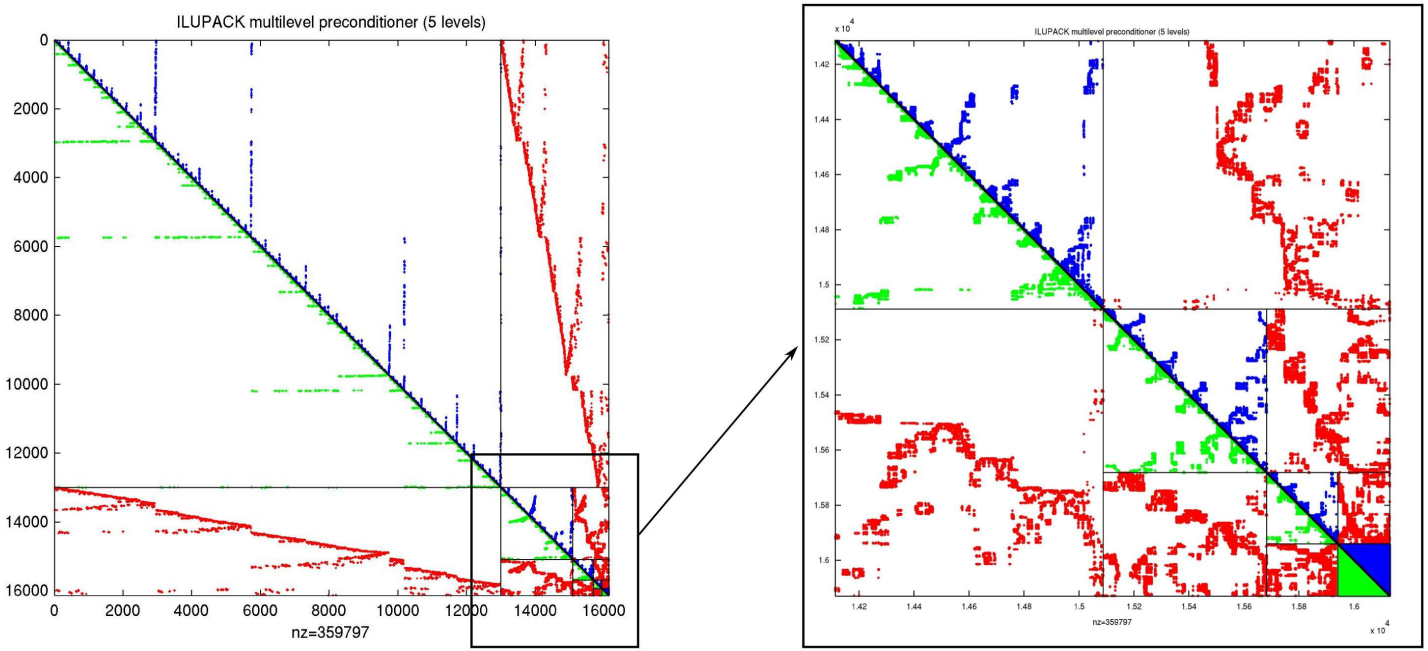
- . . . a **partial** IC decomp. of a permuted system with $\tilde{L}_B^{-1}$ satisfying $\|\tilde{L}_B^{-1}\| \lessapprox \kappa$

$$\hat{P}^T \hat{A} \hat{P} \equiv \begin{pmatrix} B & F^T \\ F & C \end{pmatrix} = \begin{pmatrix} \tilde{L}_B & 0 \\ \tilde{L}_F & I \end{pmatrix} \begin{pmatrix} \tilde{D}_B & 0 \\ 0 & \tilde{S}_c \end{pmatrix} \begin{pmatrix} \tilde{L}_B^T & \tilde{L}_F^T \\ 0 & I \end{pmatrix} + E$$

- The whole method is restarted on $\tilde{S}_c$ leading to a **multilevel** approach

Inverse-based Multilevel IC factorization

MIC factorization of five-point matrix arising from Laplace PDE discretization



$\kappa = 5$ and $\tau = 10^{-3}$ leads to a MIC factorization with 5 levels

Outline

Decomposing the problem into tasks

The key to parallelize the preconditioner computation is to decompose the IC factorization into independent tasks:

- Analyzing the sparsity pattern of A we can permute it,

$$\Pi^T A \Pi = \left(\begin{array}{cc|c} A_{11} & 0 & A_{31}^T \\ 0 & A_{22} & A_{32}^T \\ \hline A_{31} & A_{32} & A_{33} \end{array}\right).$$

$$\left[\begin{array}{ccc|ccc|c} \hat{A}_{11} & 0 & \hat{A}_{31}^T & 0 & 0 & 0 & * \\ 0 & \hat{A}_{22} & \hat{A}_{32}^T & 0 & 0 & 0 & * \\ \hat{A}_{31} & \hat{A}_{32} & \hat{A}_{33} & 0 & 0 & 0 & * \\ \hline 0 & 0 & 0 & \bar{A}_{11} & 0 & \bar{A}_{31}^T & * \\ 0 & 0 & 0 & 0 & \bar{A}_{22} & \bar{A}_{32}^T & * \\ 0 & 0 & 0 & \bar{A}_{31} & \bar{A}_{32} & \bar{A}_{33} & * \\ \hline * & * & * & * & * & * & A_{33} \end{array}\right] \rightarrow \left[\begin{array}{cccc|cc|c} A_{11} & 0 & 0 & 0 & A_{51}^T & 0 & A_{71}^T \\ 0 & A_{22} & 0 & 0 & A_{52}^T & 0 & A_{72}^T \\ 0 & 0 & A_{33} & 0 & 0 & A_{63}^T & A_{73}^T \\ 0 & 0 & 0 & A_{44} & 0 & A_{64}^T & A_{74}^T \\ \hline A_{51} & A_{52} & 0 & 0 & A_{55} & 0 & A_{75}^T \\ 0 & 0 & A_{63} & A_{64} & 0 & A_{66} & A_{76}^T \\ \hline A_{71} & A_{72} & A_{73} & A_{74} & A_{75} & A_{76} & A_{77} \end{array}\right]$$

- We can disasemble the permuted matrix into 4 submatrices

$$\left[\begin{array}{c|cc} A_{11} & A_{51}^T & A_{71}^T \\ \hline A_{51} & A_{55}^1 & A_{75}^{1\,T} \\ A_{71} & A_{75}^1 & A_{77}^1 \end{array}\right], \left[\begin{array}{c|cc} A_{22} & A_{52}^T & A_{72}^T \\ \hline A_{52} & A_{55}^2 & A_{75}^{2\,T} \\ A_{72} & A_{75}^2 & A_{77}^2 \end{array}\right], \left[\begin{array}{c|cc} A_{33} & A_{63}^T & A_{73}^T \\ \hline A_{63} & A_{66}^3 & A_{76}^{3\,T} \\ A_{73} & A_{76}^3 & A_{77}^3 \end{array}\right], \left[\begin{array}{c|cc} A_{44} & A_{64}^T & A_{74}^T \\ \hline A_{64} & A_{66}^4 & A_{76}^{4\,T} \\ A_{74} & A_{76}^4 & A_{77}^4 \end{array}\right]$$

- where

Decomposing the problem into tasks

The key to parallelize the preconditioner computation is to decompose the IC factorization into independent tasks:

- Analyzing the sparsity pattern of A we can permute it,

$$\left[\begin{array}{ccc|ccc|c} \hat{A}_{11} & 0 & \hat{A}_{31}^T & 0 & 0 & 0 & * \\ 0 & \hat{A}_{22} & \hat{A}_{32}^T & 0 & 0 & 0 & * \\ \hat{A}_{31} & \hat{A}_{32} & \hat{A}_{33} & 0 & 0 & 0 & * \\ \hline 0 & 0 & 0 & \bar{A}_{11} & 0 & \bar{A}_{31}^T & * \\ 0 & 0 & 0 & 0 & \bar{A}_{22} & \bar{A}_{32}^T & * \\ 0 & 0 & 0 & \bar{A}_{31} & \bar{A}_{32} & \bar{A}_{33} & * \\ \hline * & * & * & * & * & * & A_{33} \end{array}\right] \rightarrow \left[\begin{array}{cccc|cc|c} A_{11} & 0 & 0 & 0 & A_{51}^T & 0 & A_{71}^T \\ 0 & A_{22} & 0 & 0 & A_{52}^T & 0 & A_{72}^T \\ 0 & 0 & A_{33} & 0 & 0 & A_{63}^T & A_{73}^T \\ 0 & 0 & 0 & A_{44} & 0 & A_{64}^T & A_{74}^T \\ \hline A_{51} & A_{52} & 0 & 0 & A_{55} & 0 & A_{75}^T \\ 0 & 0 & A_{63} & A_{64} & 0 & A_{66} & A_{76}^T \\ \hline A_{71} & A_{72} & A_{73} & A_{74} & A_{75} & A_{76} & A_{77} \end{array}\right]$$

- We can disasemble the permuted matrix into 4 submatrices

$$\left[\begin{array}{c|cc} A_{11} & A_{51}^T & A_{71}^T \\ \hline A_{51} & A_{55}^1 & A_{75}^{1\,T} \\ A_{71} & A_{75}^1 & A_{77}^1 \end{array}\right], \left[\begin{array}{c|cc} A_{22} & A_{52}^T & A_{72}^T \\ \hline A_{52} & A_{55}^2 & A_{75}^{2\,T} \\ A_{72} & A_{75}^2 & A_{77}^2 \end{array}\right], \left[\begin{array}{c|cc} A_{33} & A_{63}^T & A_{73}^T \\ \hline A_{63} & A_{66}^3 & A_{76}^{3\,T} \\ A_{73} & A_{76}^3 & A_{77}^3 \end{array}\right], \left[\begin{array}{c|cc} A_{44} & A_{64}^T & A_{74}^T \\ \hline A_{64} & A_{66}^4 & A_{76}^{4\,T} \\ A_{74} & A_{76}^4 & A_{77}^4 \end{array}\right]$$

- where

$$A_{55} = A_{55}^1 + A_{55}^2\ ,\ A_{75} = A_{75}^1 + A_{75}^2 \ \mid\ A_{66} = A_{66}^3 + A_{66}^4\ ,\ A_{76} = A_{76}^3 + A_{76}^4$$

$$A_{77} = A_{77}^1 + A_{77}^2 + A_{77}^3 + A_{77}^4$$

Decomposing the problem into tasks

The key to parallelize the preconditioner computation is to decompose the IC factorization into independent tasks:

- Analyzing the sparsity pattern of A we can permute it,

$$\left[\begin{array}{ccc|ccc|c} \hat{A}_{11} & 0 & \hat{A}_{31}^T & 0 & 0 & 0 & * \\ 0 & \hat{A}_{22} & \hat{A}_{32}^T & 0 & 0 & 0 & * \\ \hat{A}_{31} & \hat{A}_{32} & \hat{A}_{33} & 0 & 0 & 0 & * \\ \hline 0 & 0 & 0 & \bar{A}_{11} & 0 & \bar{A}_{31}^T & * \\ 0 & 0 & 0 & 0 & \bar{A}_{22} & \bar{A}_{32}^T & * \\ 0 & 0 & 0 & \bar{A}_{31} & \bar{A}_{32} & \bar{A}_{33} & * \\ \hline * & * & * & * & * & * & A_{33} \end{array}\right] \rightarrow \left[\begin{array}{cccc|cc|c} A_{11} & 0 & 0 & 0 & A_{51}^T & 0 & A_{71}^T \\ 0 & A_{22} & 0 & 0 & A_{52}^T & 0 & A_{72}^T \\ 0 & 0 & A_{33} & 0 & 0 & A_{63}^T & A_{73}^T \\ 0 & 0 & 0 & A_{44} & 0 & A_{64}^T & A_{74}^T \\ \hline A_{51} & A_{52} & 0 & 0 & A_{55} & 0 & A_{75}^T \\ 0 & 0 & A_{63} & A_{64} & 0 & A_{66} & A_{76}^T \\ \hline A_{71} & A_{72} & A_{73} & A_{74} & A_{75} & A_{76} & A_{77} \end{array}\right]$$

- We can disasemble the permuted matrix into 4 submatrices

$$\left[\begin{array}{c|cc} A_{11} & A_{51}^T & A_{71}^T \\ \hline A_{51} & A_{55}^1 & A_{75}^{1\,T} \\ A_{71} & A_{75}^1 & A_{77}^1 \end{array}\right], \left[\begin{array}{c|cc} A_{22} & A_{52}^T & A_{72}^T \\ \hline A_{52} & A_{55}^2 & A_{75}^{2\,T} \\ A_{72} & A_{75}^2 & A_{77}^2 \end{array}\right], \left[\begin{array}{c|cc} A_{33} & A_{63}^T & A_{73}^T \\ \hline A_{63} & A_{66}^3 & A_{76}^{3\,T} \\ A_{73} & A_{76}^3 & A_{77}^3 \end{array}\right], \left[\begin{array}{c|cc} A_{44} & A_{64}^T & A_{74}^T \\ \hline A_{64} & A_{66}^4 & A_{76}^{4\,T} \\ A_{74} & A_{76}^4 & A_{77}^4 \end{array}\right]$$

- where

$$A_{55} = A_{55}^1 + A_{55}^2\ ,\ A_{75} = A_{75}^1 + A_{75}^2 \ \mid\ A_{66} = A_{66}^3 + A_{66}^4\ ,\ A_{76} = A_{76}^3 + A_{76}^4$$

$$A_{77} = A_{77}^1 + A_{77}^2 + A_{77}^3 + A_{77}^4$$

Decomposing the problem into tasks

The key to parallelize the preconditioner computation is to decompose the IC factorization into independent tasks:

- Analyzing the sparsity pattern of A we can permute it,

$$\left[\begin{array}{ccc|ccc|c} \hat{A}_{11} & 0 & \hat{A}_{31}^T & 0 & 0 & 0 & * \\ 0 & \hat{A}_{22} & \hat{A}_{32}^T & 0 & 0 & 0 & * \\ \hat{A}_{31} & \hat{A}_{32} & \hat{A}_{33} & 0 & 0 & 0 & * \\ \hline 0 & 0 & 0 & \bar{A}_{11} & 0 & \bar{A}_{31}^T & * \\ 0 & 0 & 0 & 0 & \bar{A}_{22} & \bar{A}_{32}^T & * \\ 0 & 0 & 0 & \bar{A}_{31} & \bar{A}_{32} & \bar{A}_{33} & * \\ \hline * & * & * & * & * & * & A_{33} \end{array}\right] \rightarrow \left[\begin{array}{cccc|cc|c} A_{11} & 0 & 0 & 0 & A_{51}^T & 0 & A_{71}^T \\ 0 & A_{22} & 0 & 0 & A_{52}^T & 0 & A_{72}^T \\ 0 & 0 & A_{33} & 0 & 0 & A_{63}^T & A_{73}^T \\ 0 & 0 & 0 & A_{44} & 0 & A_{64}^T & A_{74}^T \\ \hline A_{51} & A_{52} & 0 & 0 & A_{55} & 0 & A_{75}^T \\ 0 & 0 & A_{63} & A_{64} & 0 & A_{66} & A_{76}^T \\ \hline A_{71} & A_{72} & A_{73} & A_{74} & A_{75} & A_{76} & A_{77} \end{array}\right]$$

- We can disasemble the permuted matrix into 4 submatrices

$$\left[\begin{array}{c|cc} A_{11} & A_{51}^T & A_{71}^T \\ \hline A_{51} & A_{55}^1 & A_{75}^{1\,T} \\ A_{71} & A_{75}^1 & A_{77}^1 \end{array}\right], \left[\begin{array}{c|cc} A_{22} & A_{52}^T & A_{72}^T \\ \hline A_{52} & A_{55}^2 & A_{75}^{2\,T} \\ A_{72} & A_{75}^2 & A_{77}^2 \end{array}\right], \left[\begin{array}{c|cc} A_{33} & A_{63}^T & A_{73}^T \\ \hline A_{63} & A_{66}^3 & A_{76}^{3\,T} \\ A_{73} & A_{76}^3 & A_{77}^3 \end{array}\right], \left[\begin{array}{c|cc} A_{44} & A_{64}^T & A_{74}^T \\ \hline A_{64} & A_{66}^4 & A_{76}^{4\,T} \\ A_{74} & A_{76}^4 & A_{77}^4 \end{array}\right]$$

- where

$$A_{55} = A_{55}^1 + A_{55}^2 \;,\; A_{75} = A_{75}^1 + A_{75}^2 \;\mid\; A_{66} = A_{66}^3 + A_{66}^4 \;,\; A_{76} = A_{76}^3 + A_{76}^4$$

$$A_{77} = A_{77}^1 + A_{77}^2 + A_{77}^3 + A_{77}^4$$

Decomposing the problem into tasks

The key to parallelize the preconditioner computation is to decompose the IC factorization into independent tasks:

- In parallel, we can compute the partial IC factorization of each submatrix

$$\left[\begin{array}{c|cc} A_{11} & A_{51}^T & A_{71}^T \\ \hline A_{51} & A_{55}^1 & A_{75}^{1\,T} \\ A_{71} & A_{75}^1 & A_{77}^1 \end{array}\right] = \left[\begin{array}{c|cc} L_{11} & 0 & 0 \\ \hline L_{51} & \mathrm{I} & 0 \\ L_{71} & 0 & \mathrm{I} \end{array}\right] \left[\begin{array}{c|cc} D_{11} & 0 & 0 \\ \hline 0 & S_{55}^1 & S_{75}^{1\,T} \\ 0 & S_{75}^1 & S_{77}^1 \end{array}\right] \left[\begin{array}{c|cc} L_{11}^T & L_{51}^T & L_{71}^T \\ \hline 0 & \mathrm{I} & 0 \\ 0 & 0 & \mathrm{I} \end{array}\right].$$

- A partial IC factorization of matrix A is computed:

$$\left[\begin{array}{cccc|ccc} L_{11} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & L_{22} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & L_{33} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & L_{44} & 0 & 0 & 0 \\ \hline L_{51} & L_{52} & 0 & 0 & \mathrm{I} & 0 & 0 \\ 0 & 0 & L_{63} & L_{64} & 0 & \mathrm{I} & 0 \\ \hline L_{71} & L_{72} & L_{73} & L_{74} & 0 & 0 & \mathrm{I} \end{array}\right] \left[\begin{array}{cccc|ccc} D_{11} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & D_{22} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & D_{33} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & D_{44} & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & S_{55} & 0 & S_{75}^T \\ 0 & 0 & 0 & 0 & 0 & S_{66} & S_{76}^T \\ \hline 0 & 0 & 0 & 0 & S_{75} & S_{76} & S_{77} \end{array}\right] \left[\begin{array}{cccc|ccc} L_{11}^T & 0 & 0 & 0 & L_{51}^T & 0 & L_{71}^T \\ 0 & L_{22}^T & 0 & 0 & L_{52}^T & 0 & L_{72}^T \\ 0 & 0 & L_{33}^T & 0 & 0 & L_{63}^T & L_{73}^T \\ 0 & 0 & 0 & L_{44}^T & 0 & L_{64}^T & L_{74}^T \\ \hline 0 & 0 & 0 & 0 & \mathrm{I} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \mathrm{I} & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & \mathrm{I} \end{array}\right]$$

and restarting, we obtain a "structural" multilevel factorization.

Decomposing the problem into tasks

The key to parallelize the preconditioner computation is to decompose the IC factorization into independent tasks:

- In parallel, we can compute the partial IC factorization of each submatrix

$$\left[\begin{array}{c|cc} A_{11} & A_{51}^T & A_{71}^T \\ \hline A_{51} & A_{55}^1 & A_{75}^{1\,T} \\ A_{71} & A_{75}^1 & A_{77}^1 \end{array}\right] = \left[\begin{array}{c|cc} L_{11} & 0 & 0 \\ \hline L_{51} & \mathrm{I} & 0 \\ L_{71} & 0 & \mathrm{I} \end{array}\right] \left[\begin{array}{c|cc} D_{11} & 0 & 0 \\ \hline 0 & S_{55}^1 & S_{75}^{1\,T} \\ 0 & S_{75}^1 & S_{77}^1 \end{array}\right] \left[\begin{array}{c|cc} L_{11}^T & L_{51}^T & L_{71}^T \\ \hline 0 & \mathrm{I} & 0 \\ 0 & 0 & \mathrm{I} \end{array}\right].$$

- The Schur complements must be properly acumulated to continue

$$\left[\begin{array}{c|c} S_{55}^1 & S_{75}^{1\,T} \\ \hline S_{75}^1 & S_{77}^1 \end{array}\right] + \left[\begin{array}{c|c} S_{55}^2 & S_{75}^{2\,T} \\ \hline S_{75}^2 & S_{77}^2 \end{array}\right] = \left[\begin{array}{c|c} S_{55} & S_{75}^T \\ \hline S_{75} & S_{77}^{12} \end{array}\right]$$

$$\left[\begin{array}{c|c} S_{66}^3 & S_{76}^{3\,T} \\ \hline S_{76}^3 & S_{77}^3 \end{array}\right] + \left[\begin{array}{c|c} S_{66}^4 & S_{76}^{4\,T} \\ \hline S_{76}^4 & S_{77}^4 \end{array}\right] = \left[\begin{array}{c|c} S_{66} & S_{76}^T \\ \hline S_{76} & S_{77}^{34} \end{array}\right]$$

- Assembling two submatrices we obtain

$$\left[\begin{array}{cc|c} S_{55} & 0 & S_{75}^T \\ 0 & S_{66} & S_{76}^T \\ \hline S_{75} & S_{76} & S_{77} \end{array}\right], \quad S_{77} = S_{77}^{12} + S_{77}^{34}$$

- A partial IC factorization of matrix A is computed:

Decomposing the problem into tasks

The key to parallelize the preconditioner computation is to decompose the IC factorization into independent tasks:

- In parallel, we can compute the partial IC factorization of each submatrix

$$\left[\begin{array}{c|cc} A_{11} & A_{51}^T & A_{71}^T \\ \hline A_{51} & A_{55}^1 & A_{75}^{1\,T} \\ A_{71} & A_{75}^1 & A_{77}^1 \end{array}\right] = \left[\begin{array}{c|cc} L_{11} & 0 & 0 \\ \hline L_{51} & \mathrm{I} & 0 \\ L_{71} & 0 & \mathrm{I} \end{array}\right] \left[\begin{array}{c|cc} D_{11} & 0 & 0 \\ \hline 0 & S_{55}^1 & S_{75}^{1\,T} \\ 0 & S_{75}^1 & S_{77}^1 \end{array}\right] \left[\begin{array}{c|cc} L_{11}^T & L_{51}^T & L_{71}^T \\ \hline 0 & \mathrm{I} & 0 \\ 0 & 0 & \mathrm{I} \end{array}\right].$$

- The Schur complements must be properly acumulated to continue

$$\left[\begin{array}{c|c} S_{55}^1 & S_{75}^{1\,T} \\ \hline S_{75}^1 & S_{77}^1 \end{array}\right] + \left[\begin{array}{c|c} S_{55}^2 & S_{75}^{2\,T} \\ \hline S_{75}^2 & S_{77}^2 \end{array}\right] = \left[\begin{array}{c|c} S_{55} & S_{75}^T \\ \hline S_{75} & S_{77}^{12} \end{array}\right]$$

$$\left[\begin{array}{c|c} S_{66}^3 & S_{76}^{3\,T} \\ \hline S_{76}^3 & S_{77}^3 \end{array}\right] + \left[\begin{array}{c|c} S_{66}^4 & S_{76}^{4\,T} \\ \hline S_{76}^4 & S_{77}^4 \end{array}\right] = \left[\begin{array}{c|c} S_{66} & S_{76}^T \\ \hline S_{76} & S_{77}^{34} \end{array}\right]$$

- Assembling two submatrices we obtain

$$\left[\begin{array}{cc|c} S_{55} & 0 & S_{75}^T \\ 0 & S_{66} & S_{76}^T \\ \hline S_{75} & S_{76} & S_{77} \end{array}\right], \quad S_{77} = S_{77}^{12} + S_{77}^{34}$$

- A partial IC factorization of matrix A is computed:

Decomposing the problem into tasks

The key to parallelize the preconditioner computation is to decompose the IC factorization into independent tasks:

- In parallel, we can compute the partial IC factorization of each submatrix

$$\left[\begin{array}{c|cc} A_{11} & A_{51}^T & A_{71}^T \\ \hline A_{51} & A_{55}^1 & A_{75}^{1\,T} \\ A_{71} & A_{75}^1 & A_{77}^1 \end{array}\right] = \left[\begin{array}{c|cc} L_{11} & 0 & 0 \\ \hline L_{51} & \mathrm{I} & 0 \\ L_{71} & 0 & \mathrm{I} \end{array}\right] \left[\begin{array}{c|cc} D_{11} & 0 & 0 \\ \hline 0 & S_{55}^1 & S_{75}^{1\,T} \\ 0 & S_{75}^1 & S_{77}^1 \end{array}\right] \left[\begin{array}{c|cc} L_{11}^T & L_{51}^T & L_{71}^T \\ \hline 0 & \mathrm{I} & 0 \\ 0 & 0 & \mathrm{I} \end{array}\right].$$

- A partial IC factorization of matrix A is computed:

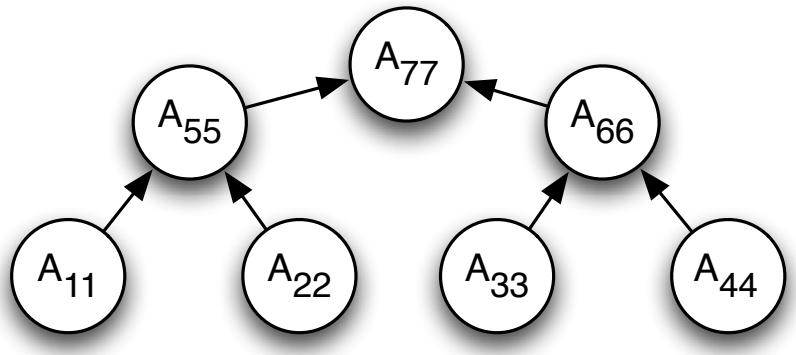
$$\left[\begin{array}{cccc|cc|c} L_{11} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & L_{22} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & L_{33} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & L_{44} & 0 & 0 & 0 \\ \hline L_{51} & L_{52} & 0 & 0 & \mathrm{I} & 0 & 0 \\ 0 & 0 & L_{63} & L_{64} & 0 & \mathrm{I} & 0 \\ \hline L_{71} & L_{72} & L_{73} & L_{74} & 0 & 0 & \mathrm{I} \end{array}\right] \left[\begin{array}{cccc|cc|c} D_{11} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & D_{22} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & D_{33} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & D_{44} & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & S_{55} & 0 & S_{75}^T \\ 0 & 0 & 0 & 0 & 0 & S_{66} & S_{76}^T \\ \hline 0 & 0 & 0 & 0 & S_{75} & S_{76} & S_{77} \end{array}\right] \left[\begin{array}{cccc|cc|c} L_{11}^T & 0 & 0 & 0 & L_{51}^T & 0 & L_{71}^T \\ 0 & L_{22}^T & 0 & 0 & L_{52}^T & 0 & L_{72}^T \\ 0 & 0 & L_{33}^T & 0 & 0 & L_{63}^T & L_{73}^T \\ 0 & 0 & 0 & L_{44}^T & 0 & L_{64}^T & L_{74}^T \\ \hline 0 & 0 & 0 & 0 & \mathrm{I} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \mathrm{I} & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & \mathrm{I} \end{array}\right]$$

and restarting, we obtain a "structural" multilevel factorization.

Decomposing the problem into tasks

The key to parallelize the preconditioner computation is to decompose the IC factorization into independent tasks:

- The task dependency tree represents the behaviour of the computation

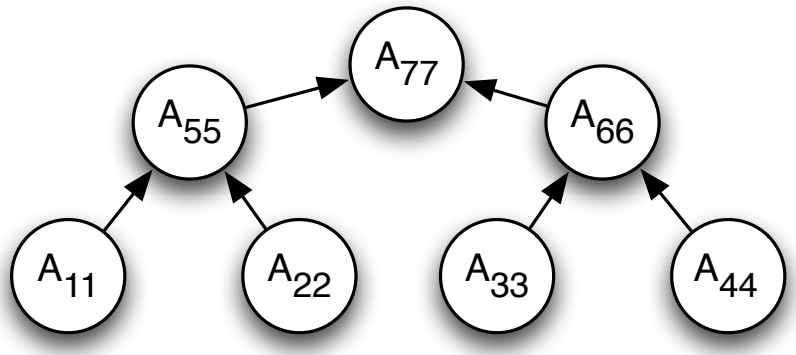


- Each node refers to a block to be factorized, while the edges define the data dependency.
- The nodes without any dependency can be factorized in parallel.
- The non-leaf nodes have to wait until its children have finalized.

Decomposing the problem into tasks

The key to parallelize the preconditioner computation is to decompose the IC factorization into independent tasks:

- The task dependency tree represents the behaviour of the computation

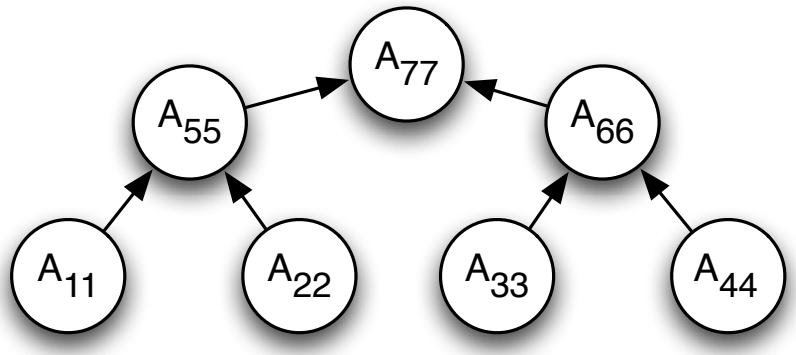


- Each node refers to a block to be factorized, while the edges define the data dependency.
- The nodes without any dependency can be factorized in parallel.
- The non-leaf nodes have to wait until its children have finalized.

Decomposing the problem into tasks

The key to parallelize the preconditioner computation is to decompose the IC factorization into independent tasks:

- The task dependency tree represents the behaviour of the computation

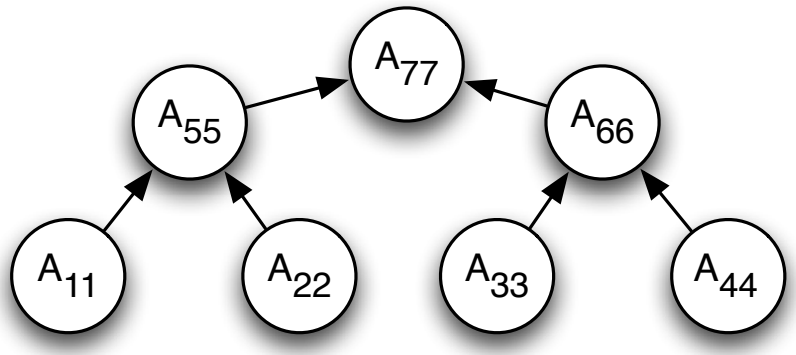


- Each node refers to a block to be factorized, while the edges define the data dependency.
- The nodes without any dependency can be factorized in parallel.
- The non-leaf nodes have to wait until its children have finalized.

Decomposing the problem into tasks

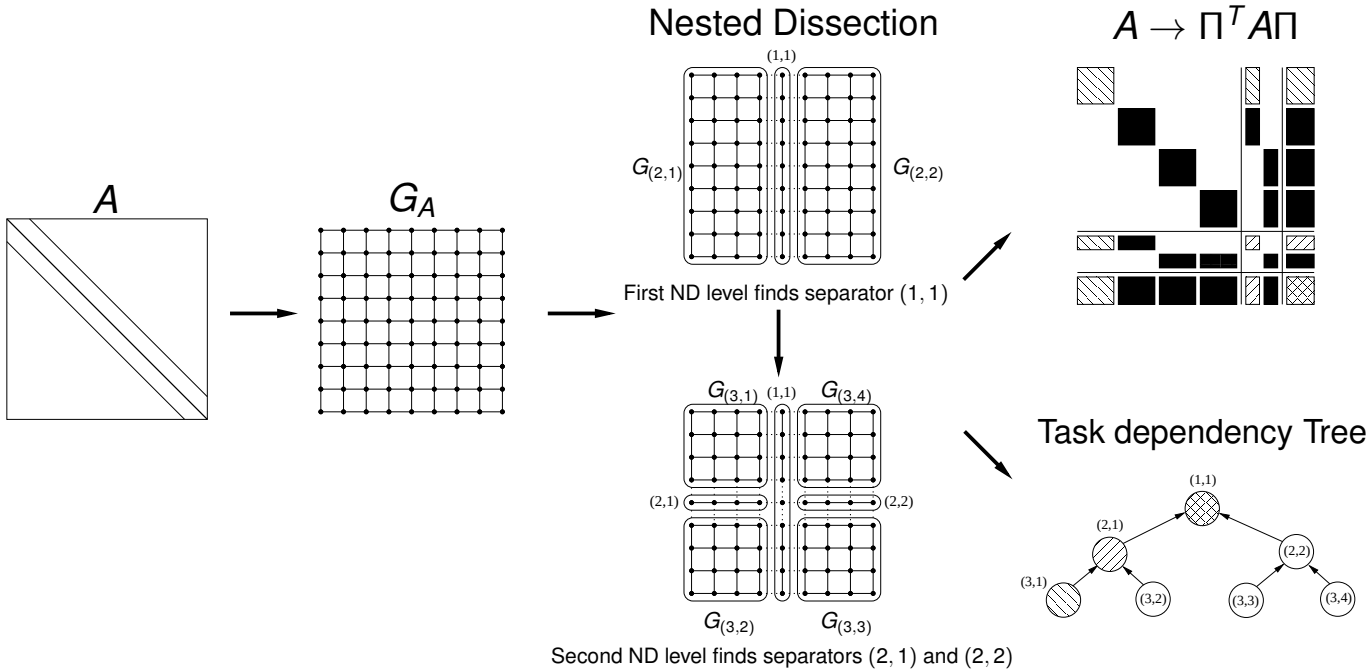
The key to parallelize the preconditioner computation is to decompose the IC factorization into independent tasks:

- The task dependency tree represents the behaviour of the computation



- Each node refers to a block to be factorized, while the edges define the data dependency.
- The nodes without any dependency can be factorized in parallel.
- The non-leaf nodes have to wait until its children have finalized.

Decomposing the problem into tasks



- Task-level parallelism revealed by nested dissection on adjacency graph G_A
  $\rightarrow$ fast and efficient **multilevel** (MLND) variants in METIS and SCOTCH
- Parallelization driven by the dependencies "captured" in the task tree
  $\rightarrow$ subdomains **first** eliminated independently, **then** separators $(2, 1)$ and $(2, 2)$ in parallel, **finally** the root separator $(3, 1)$

Parallel Inverse-based Multilevel IC factorization

The parallelization of ILUPACK must consider two multilevel procedures

- ILUPACK computes Multilevel IC factorization
  $\rightarrow$ Numerical levels related to inverse-based IC factorizations
- Parallelization of IC factorization
  $\rightarrow$ Structural levels computed by nested disection algorithm

The solution is to combine both multilevel procedures

- Structural levels are initially used to reveal the concurrence
- Numerical levels appears during the MIC factorization of each submatrix

The Multilevel IC factorization must be adjusted when it is applied on a submatrix

- Where to move the rejected pivots
- When to stop the factorization of a submatrix

Parallel Inverse-based Multilevel IC factorization

The parallelization of ILUPACK must consider two multilevel procedures

- ILUPACK computes Multilevel IC factorization
  $\rightarrow$ Numerical levels related to inverse-based IC factorizations
- Parallelization of IC factorization
  $\rightarrow$ Structural levels computed by nested disection algorithm

The solution is to combine both multilevel procedures

- Structural levels are initially used to reveal the concurrence
- Numerical levels appears during the MIC factorization of each submatrix

The Multilevel IC factorization must be adjusted when it is applied on a submatrix

- Where to move the rejected pivots
- When to stop the factorization of a submatrix

Parallel Inverse-based Multilevel IC factorization

The parallelization of ILUPACK must consider two multilevel procedures

- ILUPACK computes Multilevel IC factorization
  $\rightarrow$ Numerical levels related to inverse-based IC factorizations
- Parallelization of IC factorization
  $\rightarrow$ Structural levels computed by nested disection algorithm

The solution is to combine both multilevel procedures

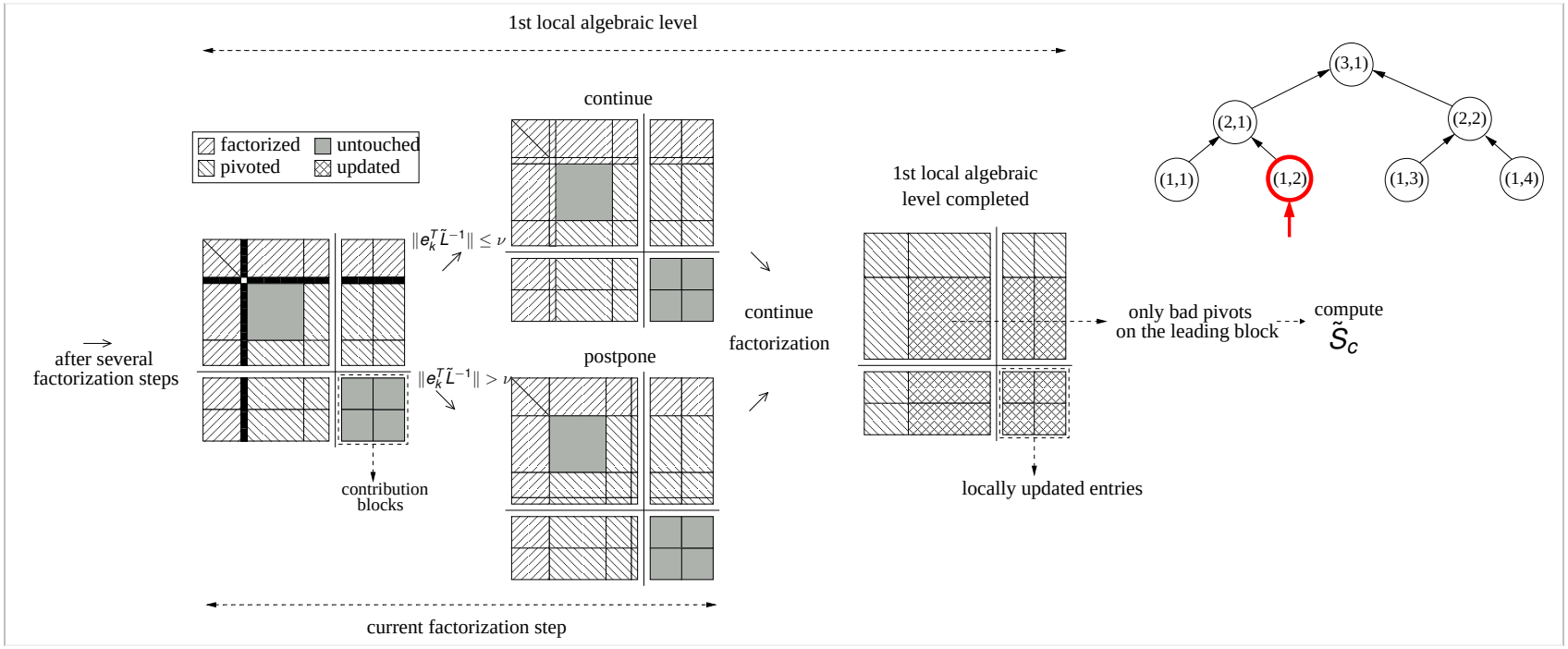
- Structural levels are initially used to reveal the concurrence
- Numerical levels appears during the MIC factorization of each submatrix

The Multilevel IC factorization must be adjusted when it is applied on a submatrix

- Where to move the rejected pivots
- When to stop the factorization of a submatrix

Parallel Inverse-based Multilevel IC factorization

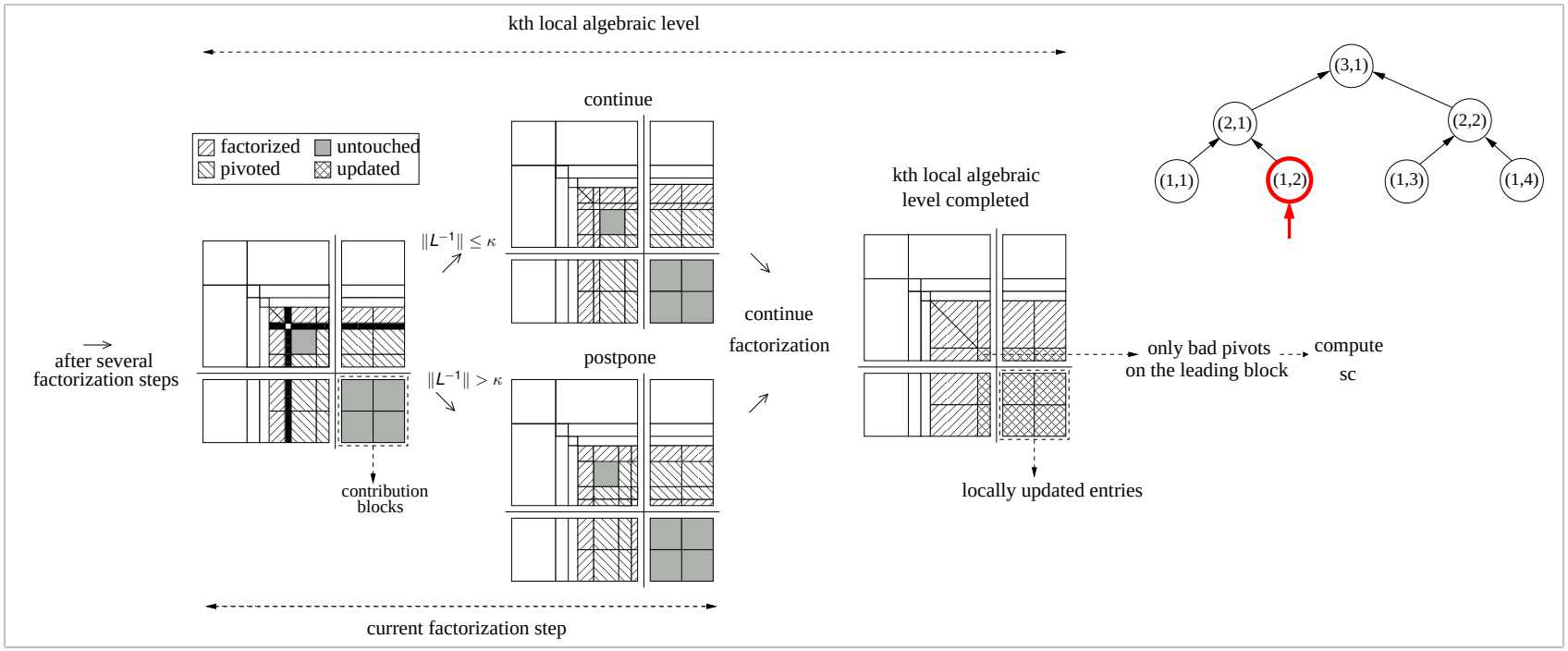
The tasks compute a local MIC **restricted to the leading block** of the submatrix



- IC decomp. employs **restricted** Inverse-based pivoting
  → Rejected pivots **are only moved** to the bottom/right-end of the leading block
- IC decomp. stops if the trailing of the **leading block** only contains rejected pivots
  → Rejected pivots still “resolved” inside the task by entering next MIC level

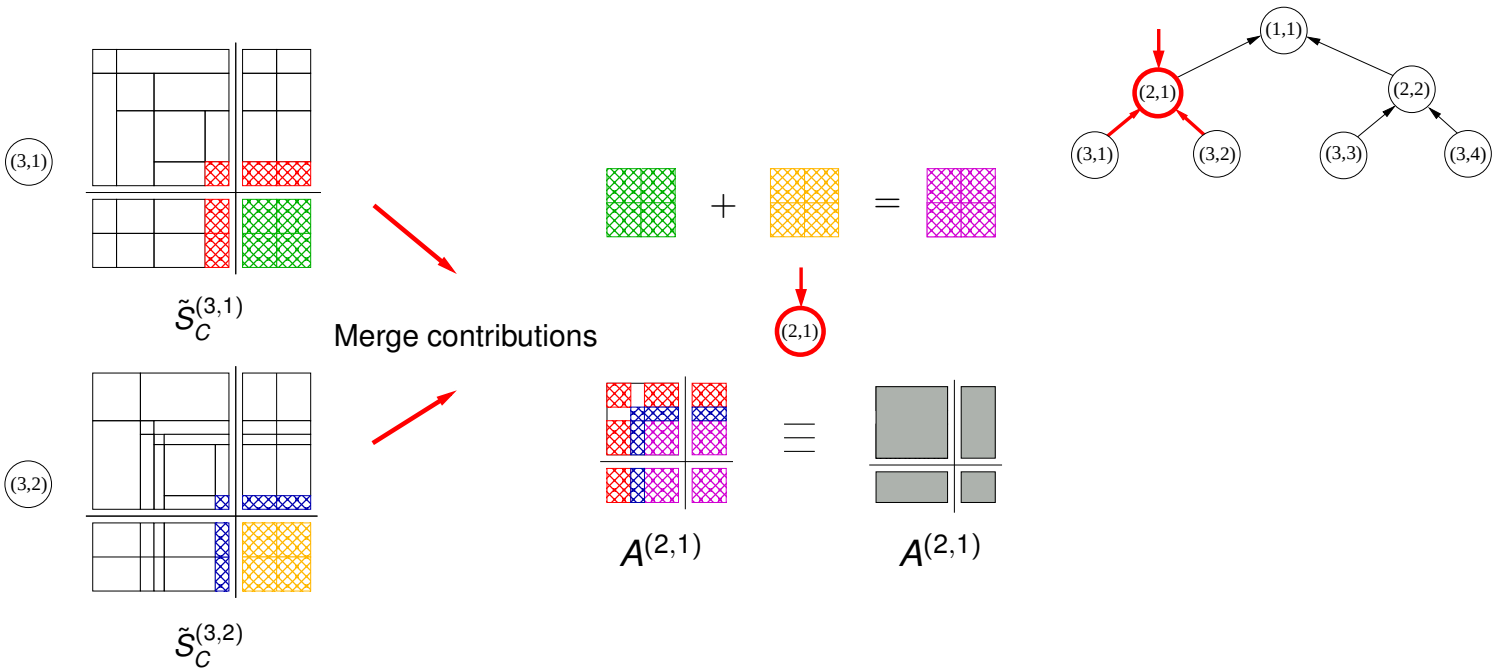
Parallel Inverse-based Multilevel IC factorization

The tasks compute a local MIC **restricted to the leading block** of the submatrix

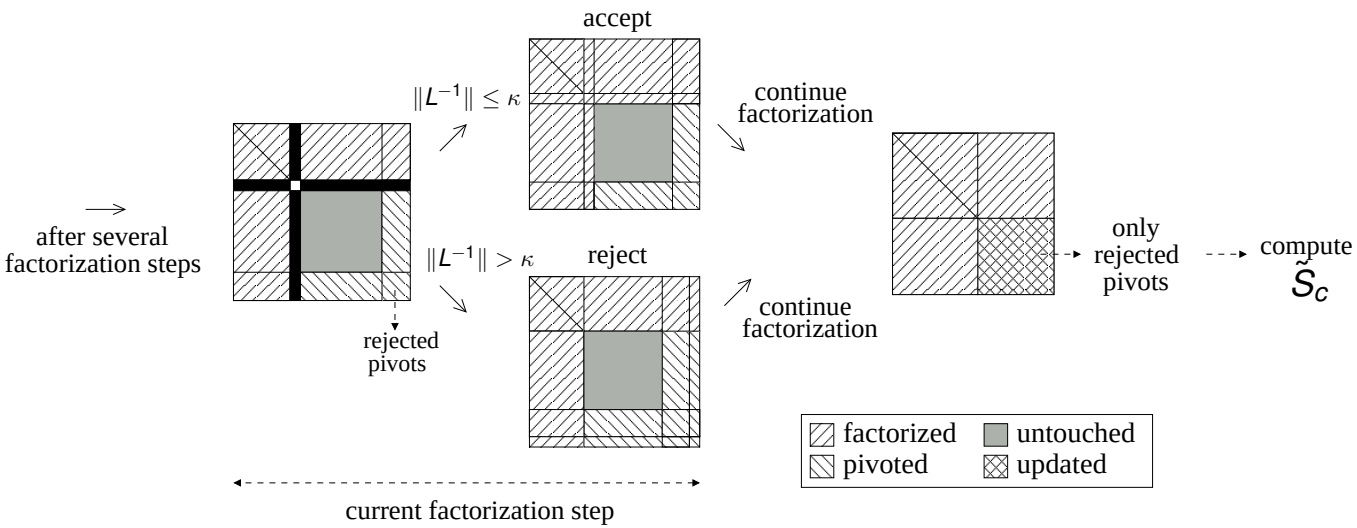


- When the set of rejected pivots is “small enough” the local MIC **stops**
  → The task “sends” the local contributions to the parent task

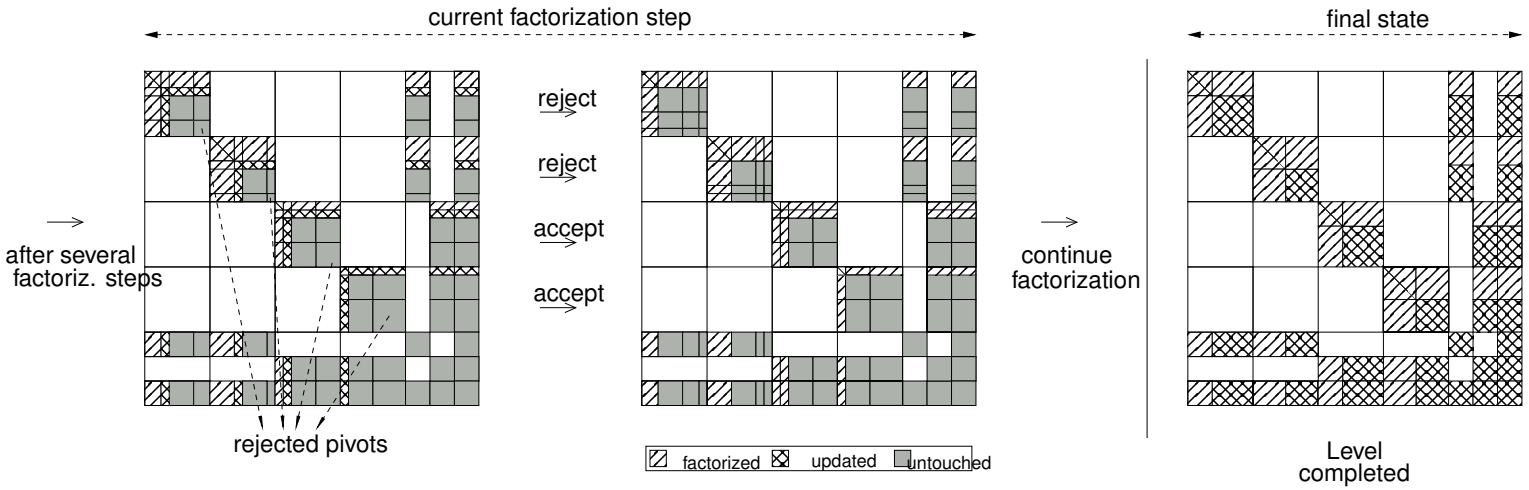
Parallel inverse-based MIC factorization



- The parent task merges contributions resulting from its children
- A new submatrix is constructed for the local MIC decomp. of the separator task
  - $\rightarrow$ Bad pivots resulting from its children copied on top of the leading block
  - $\rightarrow$ Contributions blocks added



inverse-based Multilevel IC factorization

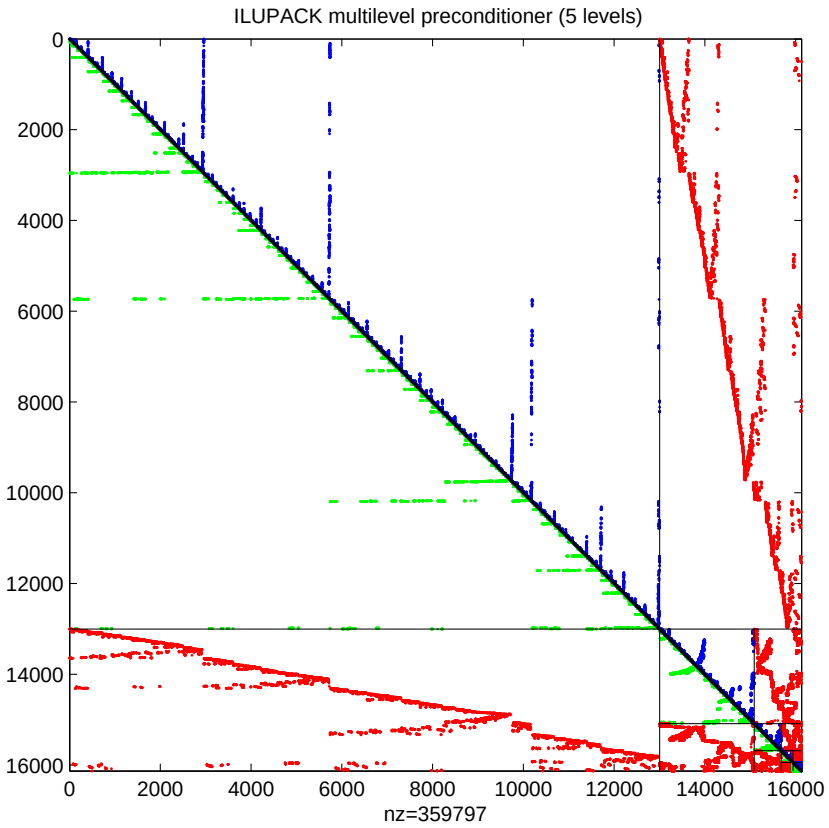


Parallel inverse-based Multilevel IC factorization

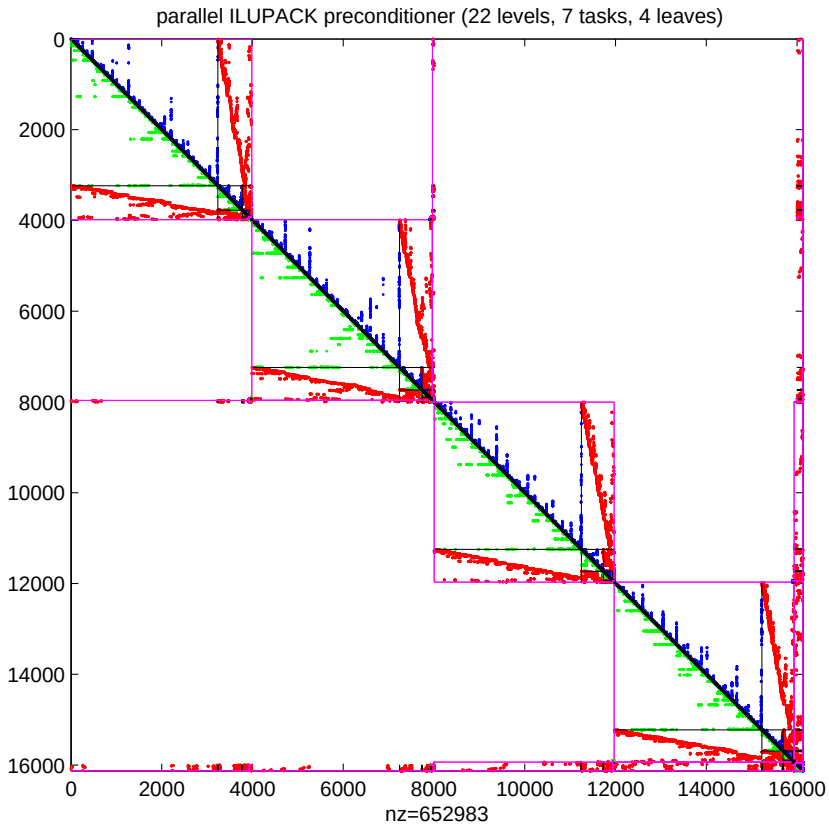
Parallel vs Sequential Inverse-based MIC factoriz.

Factorization of five-point matrix arising from Laplace PDE discretization

MIC (5 levels)



Parallel MIC (4 leaves, 22 levels)

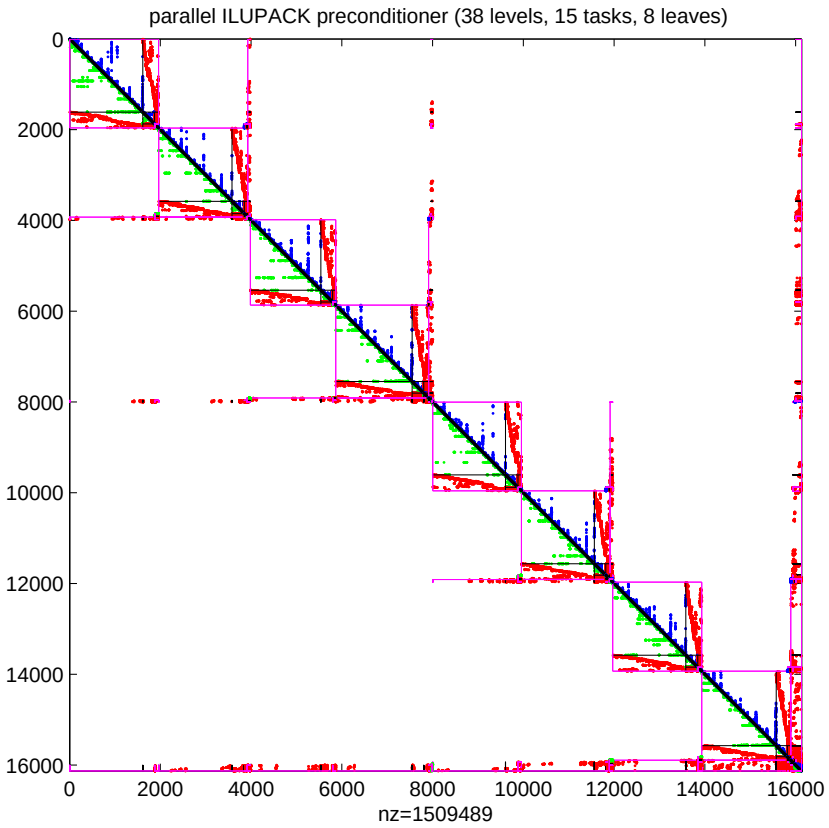


$\kappa = 5$ and $\tau = 10^{-3}$

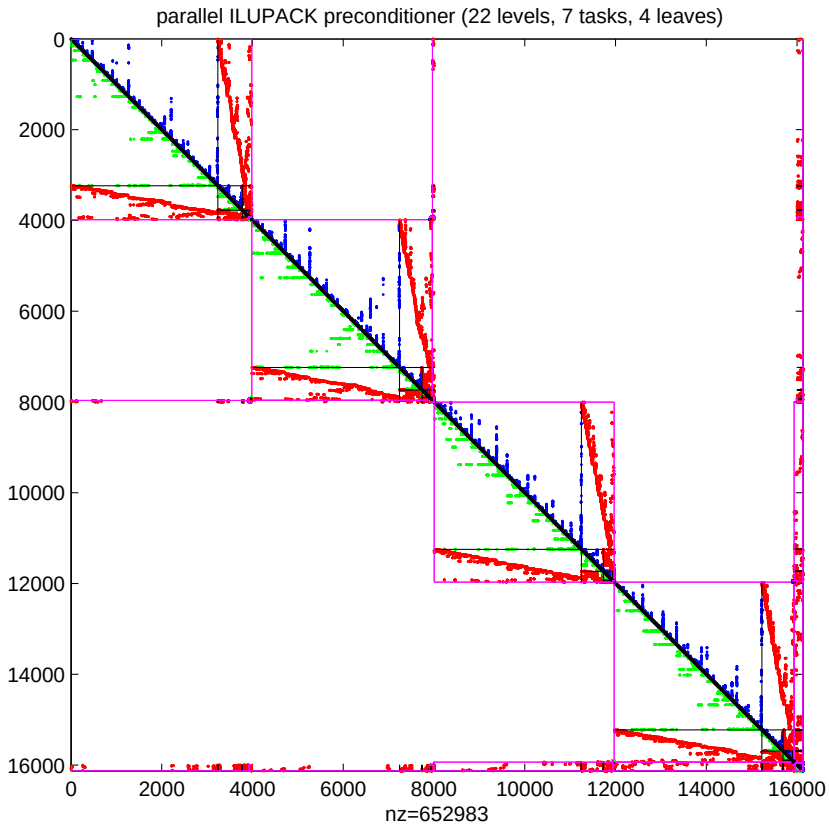
Parallel vs Sequential Inverse-based MIC factoriz.

Factorization of five-point matrix arising from Laplace PDE discretization

Parallel MIC (8 leaves, 38 levels)



Parallel MIC (4 leaves, 22 levels)



$\kappa = 5$ and $\tau = 10^{-3}$

OmpSs programming model

On the way to Exascale, one of the main challenges is the parallel programming revolution:

- In the classical programming models, the code defines
  - Where to place data
  - What to run and where
  - How to communicate
- In the new programming models, the questions have changed,
  - What do I need to compute?
  - What data do I need to use?
  - Hints (not necessarily very precise) on potential concurrency, locality, ...
- For the programmers, the main feature of the new programming models is to write a single source useful for any target (multicores, multicomputers, heterogeneous systems with GPU?s, ...)

OmpSs programming model

On the way to Exascale, one of the main challenges is the parallel programming revolution:

- In the classical programming models, the code defines
  - Where to place data
  - What to run and where
  - How to communicate
- In the new programming models, the questions have changed,
  - What do I need to compute?
  - What data do I need to use?
  - Hints (not necessarily very precise) on potential concurrency, locality, ...
- For the programmers, the main feature of the new programming models is to write a single source useful for any target (multicores, multicomputers, heterogeneous systems with GPU?s, ...)

OmpSs programming model

On the way to Exascale, one of the main challenges is the parallel programming revolution:

- In the classical programming models, the code defines
  - Where to place data
  - What to run and where
  - How to communicate
- In the new programming models, the questions have changed,
  - What do I need to compute?
  - What data do I need to use?
  - Hints (not necessarily very precise) on potential concurrency, locality, ...
- For the programmers, the main feature of the new programming models is to write a single source useful for any target (multicores, multicomputers, heterogeneous systems with GPU?s, ...)

OmpSs programming model

OmpSs is a new programming model, which extends OpenMP:

- Define a single address space
- The code can spawn tasks that will be executed in the future
- Tasks annotated with directionality clauses,

```
#pragma omp task in(lvalue_expr-list) \
                 out(lvalue_expr-list) \
                 inout(lvalue_expr-list)
```

  used to build dependences between tasks.
- The tasks wait to data to be produced by other tasks (in clause).
- Basis for memory management functionalities (replication, locality, movement, ...)

OmpSs programming model

OmpSs is a new programming model, which extends OpenMP:

- Define a single address space
- The code can spawn tasks that will be executed in the future
- Tasks annotated with directionality clauses,

```
#pragma omp task in(lvalue_expr-list) \
                 out(lvalue_expr-list) \
                 inout(lvalue_expr-list)
```

  used to build dependences between tasks.

- The tasks wait to data to be produced by other tasks (in clause).
- Basis for memory management functionalities (replication, locality, movement, ...)

OmpSs programming model

OmpSs is a new programming model, which extends OpenMP:

- Define a single address space
- The code can spawn tasks that will be executed in the future
- Tasks annotated with directionality clauses,

```
#pragma omp task in(lvalue_expr-list) \
                 out(lvalue_expr-list) \
                 inout(lvalue_expr-list)
```

  used to build dependences between tasks.
- The tasks wait to data to be produced by other tasks (in clause).
- Basis for memory management functionalities (replication, locality, movement, ...)

OmpSs memory model

From the point of view of the programmer a single naming space exists

- The standard sequential/shared memory address space

From the point of view of the runtime and target platform, different possible scenarios

- Pure SMP:
  - Single address space
- Distributed/heterogeneous systems (cluster, gpus, ...):
  - Multiple address spaces exists, such that versions of same data may exist in several ones.
  - Data consistency is ensured by the implementation

OmpSs memory model

From the point of view of the programmer a single naming space exists

- The standard sequential/shared memory address space

From the point of view of the runtime and target platform, different possible scenarios

- Pure SMP:
  - Single address space
- Distributed/heterogeneous systems (cluster, gpus, ...):
  - Multiple address spaces exists, such that versions of same data may exist in several ones.
  - Data consistency is ensured by the implementation

OmpSs memory model

From the point of view of the programmer a single naming space exists

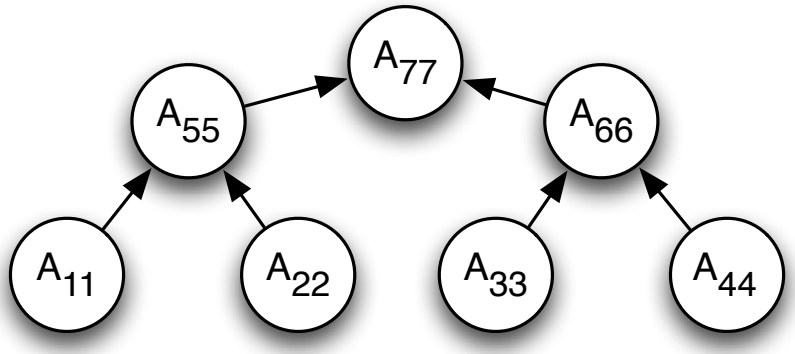
- The standard sequential/shared memory address space

From the point of view of the runtime and target platform, different possible scenarios

- Pure SMP:
  - Single address space
- Distributed/heterogeneous systems (cluster, gpus, ...):
  - Multiple address spaces exists, such that versions of same data may exist in several ones.
  - Data consistency is ensured by the implementation

Preconditioner computation on OmpSs

Task dependency tree:



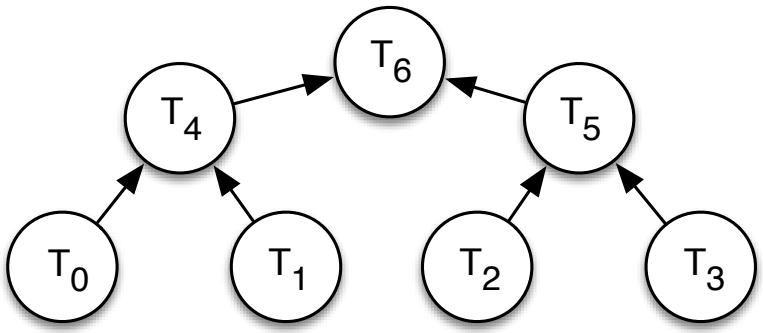
We define a structure called dag[][] to define the dependencies in OmpSs

Block		A_{11}	A_{22}	A_{33}	A_{44}	A_{55}	A_{66}	A_{77}
Task		T_0	T_1	T_2	T_3	T_4	T_5	T_6
Task id. j		0	1	2	3	4	5	6
Left descendant	dag[0][j]	–	–	–	–	0	2	4
Right descendant	dag[1][j]	–	–	–	–	1	3	5
Ancestor	dag[2][j]	4	4	5	5	6	6	–

The dag is created and initialized before the parallel computation begins

Preconditioner computation on OmpSs

Task dependency tree:



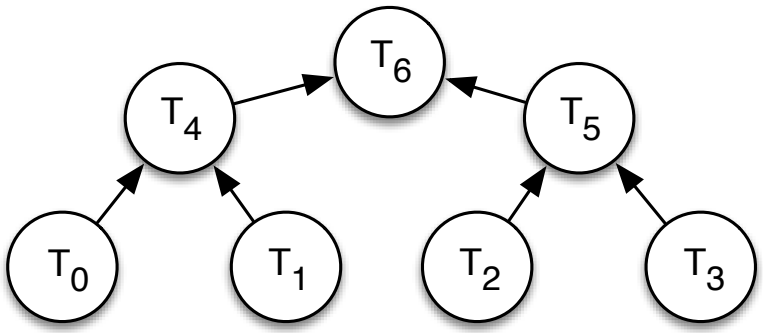
We define a structure called dag[][] to define the dependencies in OmpSs

Block		A_{11}	A_{22}	A_{33}	A_{44}	A_{55}	A_{66}	A_{77}
Task		T_0	T_1	T_2	T_3	T_4	T_5	T_6
Task id. j		0	1	2	3	4	5	6
Left descendant	`dag[0][`j`]`	–	–	–	–	0	2	4
Right descendant	`dag[1][`j`]`	–	–	–	–	1	3	5
Ancestor	`dag[2][`j`]`	4	4	5	5	6	6	–

The dag is created and initialized before the parallel computation begins

Preconditioner computation on OmpSs

Task dependency tree:



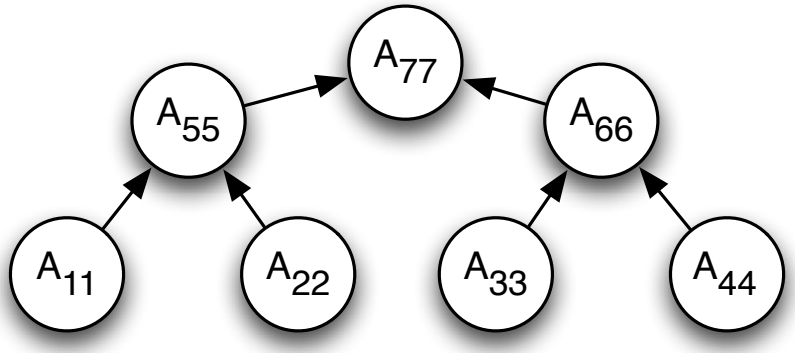
We define a structure called dag[][] to define the dependencies in OmpSs

Block		A_{11}	A_{22}	A_{33}	A_{44}	A_{55}	A_{66}	A_{77}
Task		T_0	T_1	T_2	T_3	T_4	T_5	T_6
Task id. j		0	1	2	3	4	5	6
Left descendant	`dag[0][`j`]`	0	1	2	3	0	2	4
Right descendant	`dag[1][`j`]`	0	1	2	3	1	3	5
Ancestor	`dag[2][`j`]`	4	4	5	5	6	6	6

The dag is created and initialized before the parallel computation begins

Preconditioner computation on OmpSs

Task dependency tree:

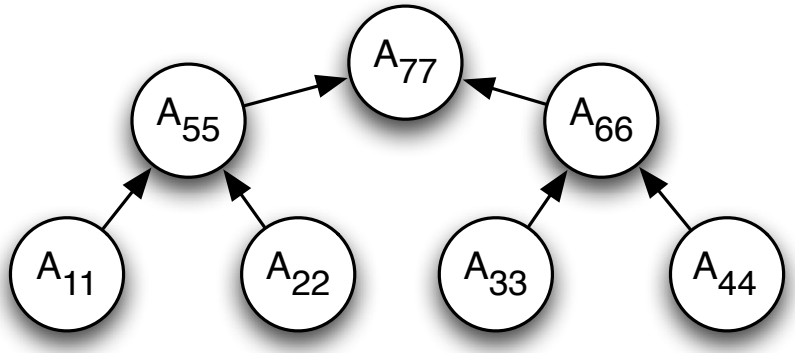


Given the complexity of the data structure, we use a vector of sentinels to define the data dependency.

```
for(i=0; i<dimL; i++){
  tsk_id=i;
  left=dag[0][tsk_id]; right=dag[1][tsk_id];
  #pragma omp task in(v_sentinel[left],v_sentinel[right]) \
                   out(v_sentinel[tsk_id])
  ILUPrecond(spMat, spFact, taskid, v_sentinel, dag);
}
```

Preconditioner computation on OmpSs

Task dependency tree:

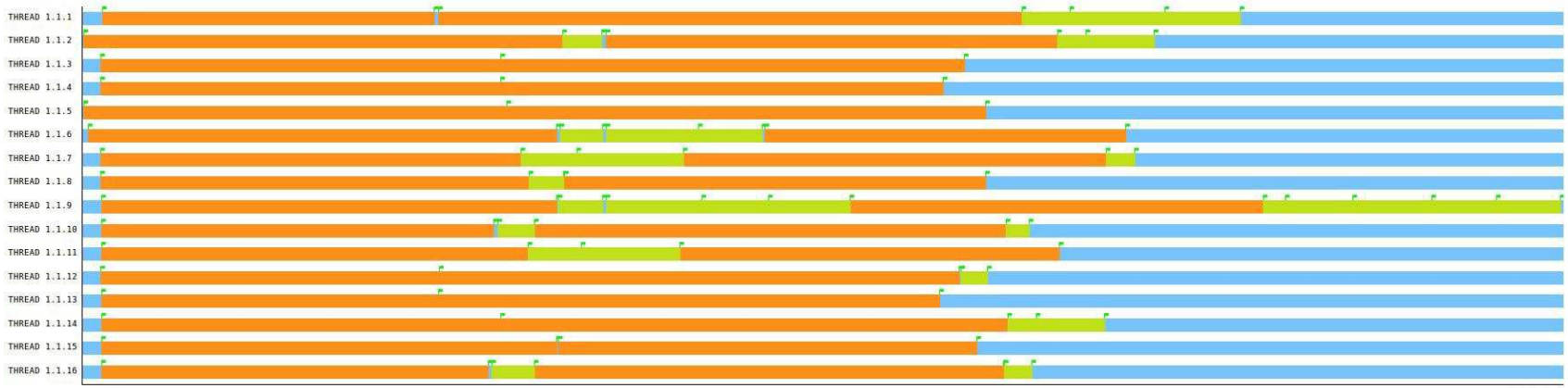


Given the complexity of the data structure, we use a vector of sentinels to define the data dependency.

```
for(i=0; i<dimL; i++){
  tsk_id=i;
  left=dag[0][tsk_id]; right=dag[1][tsk_id];
  #pragma omp task in(v_sentinel[left],v_sentinel[right]) \
                   out(v_sentinel[tsk_id])
  ILUPrecond(spMat, spFact, taskid, v_sentinel, dag);
}
```

Preconditioner computation on OmpSs

PARAVER trace of the preconditioner computation (32 leaves & 16 threads):



Analyzing the trace, we observe that:

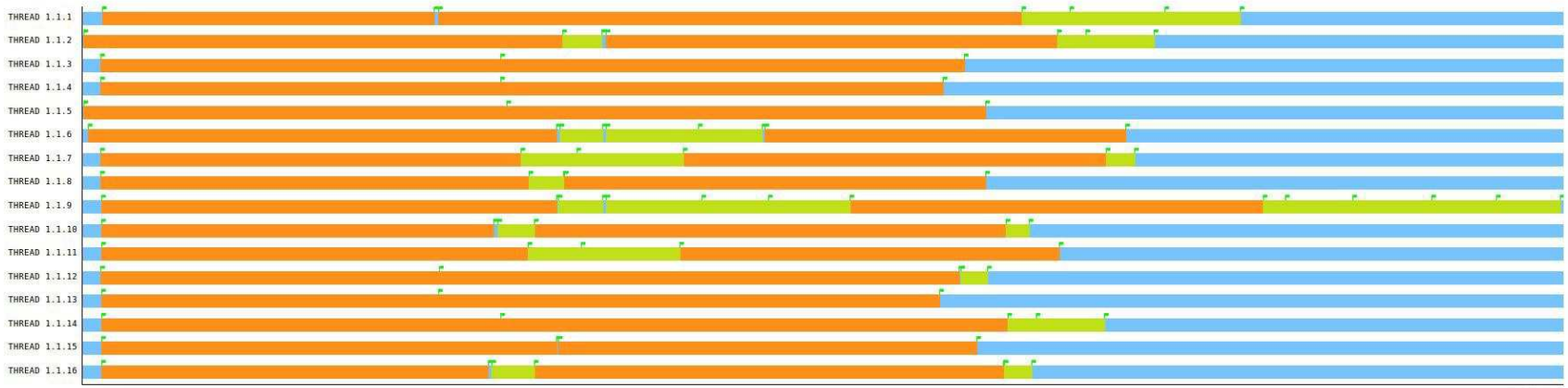
- The leaf tasks (orange) are larger than the non-leaf tasks (green)
- Some small tasks are processed before the larger tasks

Therefore, some priorities could be defined to improve performances:

- To process leaf tasks before non-leaf tasks
- ...

Preconditioner computation on OmpSs

PARAVER trace of the preconditioner computation (32 leaves & 16 threads):



Analyzing the trace, we observe that:

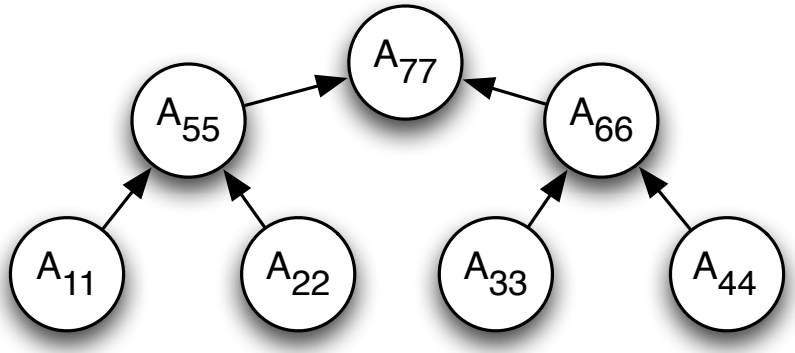
- The leaf tasks (orange) are larger than the non-leaf tasks (green)
- Some small tasks are processed before the larger tasks

Therefore, some priorities could be defined to improve performances:

- To process leaf tasks before non-leaf tasks
- ...

Preconditioner computation on OmpSs

Task dependency tree:

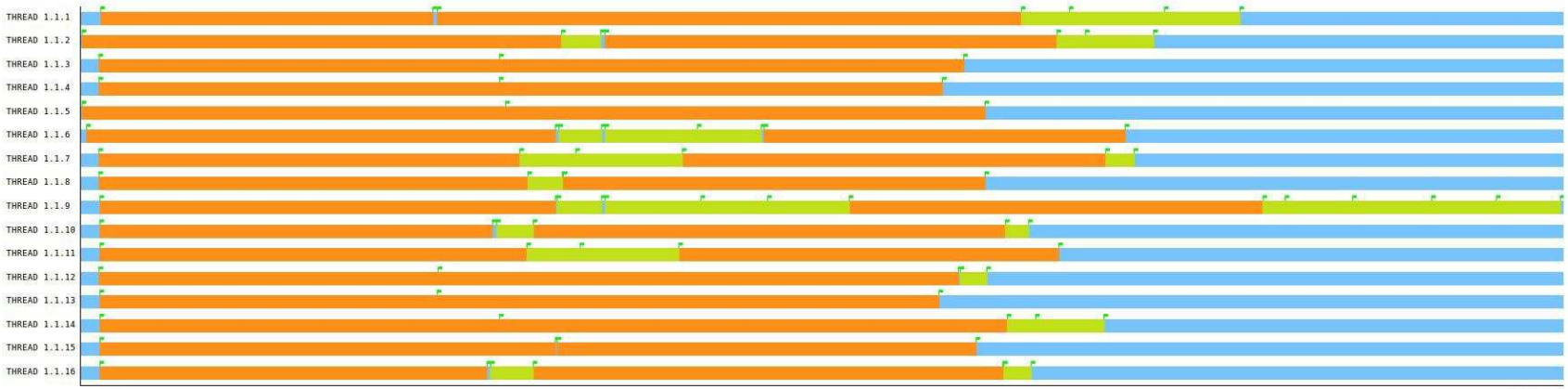


The priority could be used to improve performances

```
for(i=0; i<dimL; i++){
  tsk_id=tsk_ord[i]; prio=tsk_prio[i];
  left=dag[0][tsk_id]; right=dag[1][tsk_id];
  #pragma omp task in(v_sentinel[left],v_sentinel[right]) \
                   out(v_sentinel[tsk_id]) priority(prio)
  ILUPrecond(spMat, spFact, tsk_id, v_sentinel, dag);
}
```

Preconditioner computation on OmpSs

PARAVER trace without priority:



PARAVER trace with priority:

Outline

Experimental Framework

- Target platforms:
  - 4 AMD Opteron 6276 Processors (16 cores, total of 64):
    - 2.10 GHz
    - 64 Gbytes RAM
  - 2 Intel Xeon E5-2670 (8 cores, total of 16)
    - 2.60 GHz
    - 32 Gbytes RAM
  - All executions use up to 16 cores.
- IEEE double precision
- OmpSs version: Nanox 0.7a, mcxx 1.99

Example Benchmark

We consider the Laplacian PDE equation

$$-\Delta u = f$$

in a $3D$ unit cube $\Omega = [0,1]^3$ with Dirichlet boundary conditions $u = g$ on $\partial\Omega$

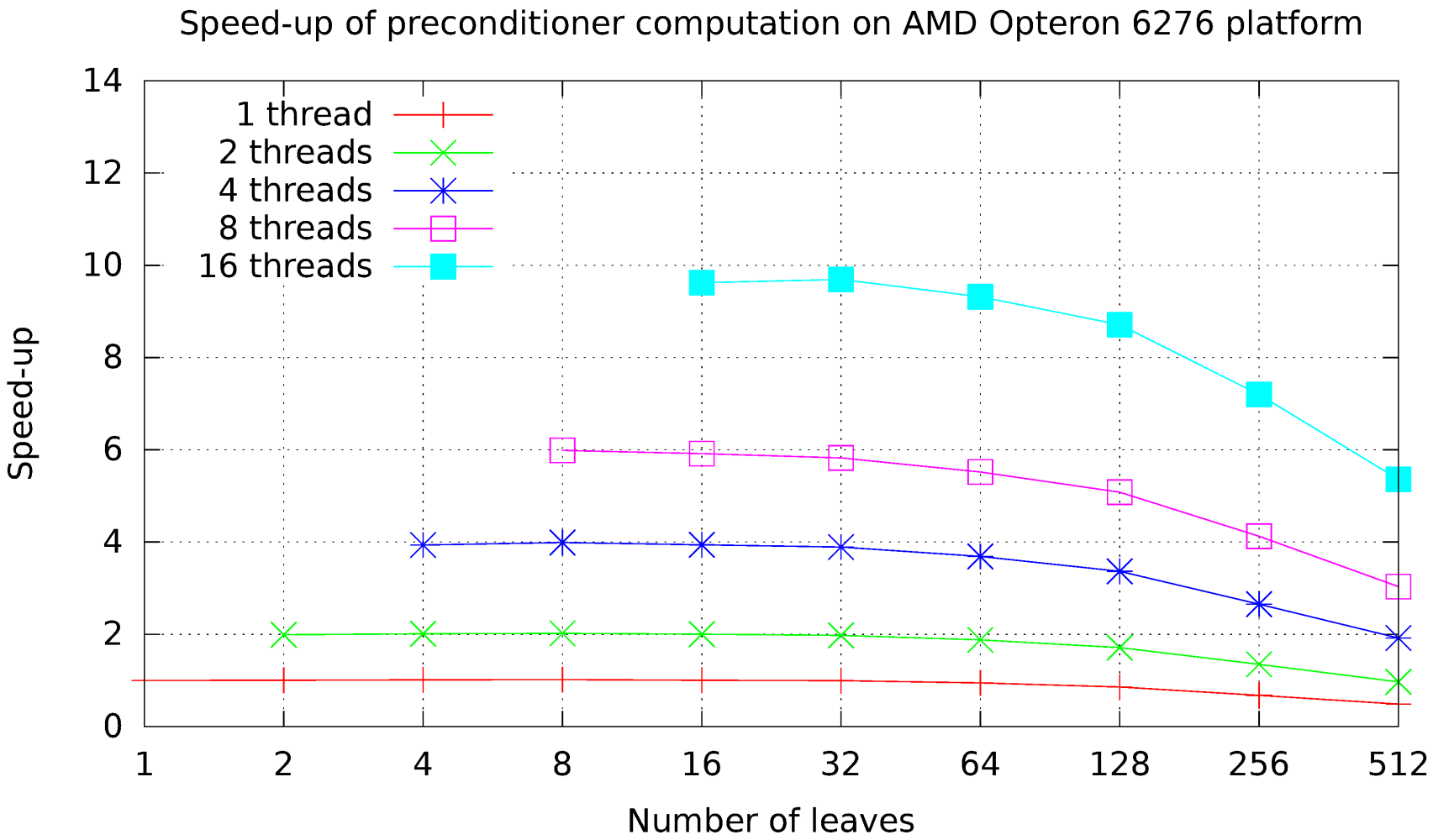
For the discretization . . .

- Ω replaced by a $N_x \times N_y \times N_z$ uniform grid
- Δu approximated by centered finite differences
  $\rightarrow$ seven-point-star difference stencil

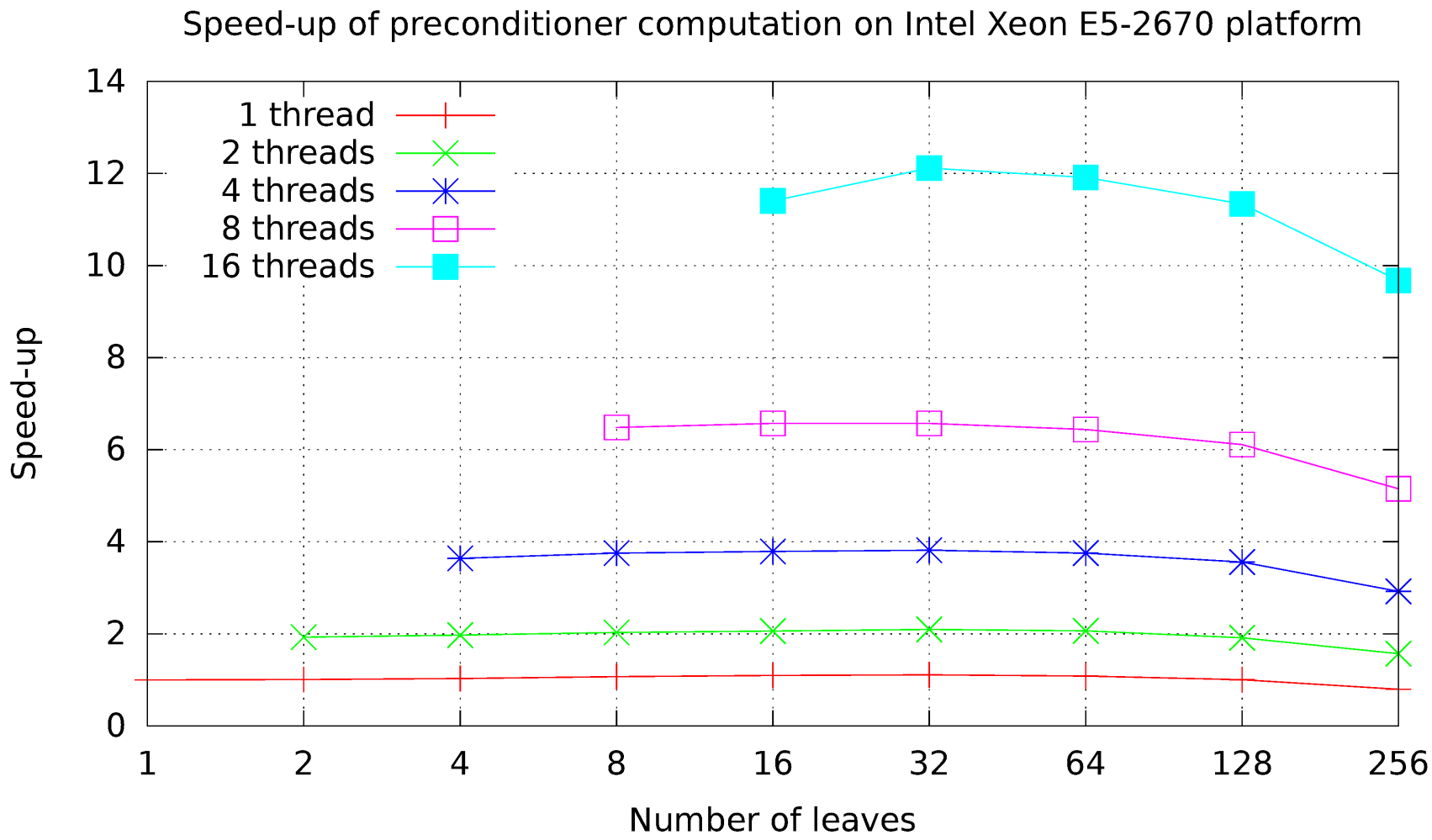
In this talk, we only consider the largest benchmark problem that fitted into the main memory of the target machines

$N_x \times N_y \times N_z$	n	nnz_A/n
$200 \times 200 \times 200$	8,000,000	7

OMPSS preconditioner speed-up in AMD



OMPSS preconditioner speed-up in INTEL



Outline

Preconditioned Conjugate Gradient

Initialize $k, x_0, r_0, z_0, d_0, \beta_0, \tau_0$
while $((k < k_{max})\&\&(\tau_k > \tau_{min}))$
$w_k := Ad_k$
$\gamma_k := d_k^T w_k$
$\rho_k := \beta_k / \gamma_k$
$x_{k+1} := x_k + \rho_k d_k$
$r_{k+1} := r_k - \rho_k w_k$
$z_{k+1} := M^{-1} r_{k+1}$
$\alpha_k := \beta_k$
$\beta_{k+1} := r_{k+1}^T z_{k+1}$
$\alpha_k := \beta_{k+1} / \alpha_k$
$d_{k+1} := z_{k+1} + \alpha_k d_k$
$\tau_{k+1} := ||r_{k+1}||_2$
$k := k + 1$
end while

Preconditioned Conjugate Gradient

Initialize $k, x_0, r_0, z_0, d_0, \beta_0, \tau_0$	
while $((k < k_{max})\&\&(\tau_k > \tau_{min}))$	
$w_k := Ad_k$	SPMV
$\gamma_k := d_k^T w_k$	DOT
$\rho_k := \beta_k / \gamma_k$	
$x_{k+1} := x_k + \rho_k d_k$	AXPY
$r_{k+1} := r_k - \rho_k w_k$	AXPY
$z_{k+1} := M^{-1} r_{k+1}$	Preconditioning
$\alpha_k := \beta_k$	
$\beta_{k+1} := r_{k+1}^T z_{k+1}$	DOT
$\alpha_k := \beta_{k+1} / \alpha_k$	
$d_{k+1} := z_{k+1} + \alpha_k d_k$	AXPY-like
$\tau_{k+1} := \|r_{k+1}\|_2$	2-norm
$k := k + 1$	
end while	

Preconditioned Conjugate Gradient

Initialize $k, x_0, r_0, z_0, d_0, \beta_0, \tau_0$
while $((k < k_{max})\&\&(\tau_k > \tau_{min}))$
$w_k := Ad_k$
$\gamma_k := d_k^T w_k$
$\rho_k := \beta_k / \gamma_k$
$x_{k+1} := x_k + \rho_k d_k$
$r_{k+1} := r_k - \rho_k w_k$
$z_{k+1} := M^{-1} r_{k+1}$
$\alpha_k := \beta_k$
$\beta_{k+1} := r_{k+1}^T z_{k+1}$
$\alpha_k := \beta_{k+1} / \alpha_k$
$d_{k+1} := z_{k+1} + \alpha_k d_k$
$\tau_{k+1} := ||r_{k+1}||_2$
$k := k + 1$
end while

SPMV $\rightarrow$ on leaves
DOT $\rightarrow$ on leaves + reduction

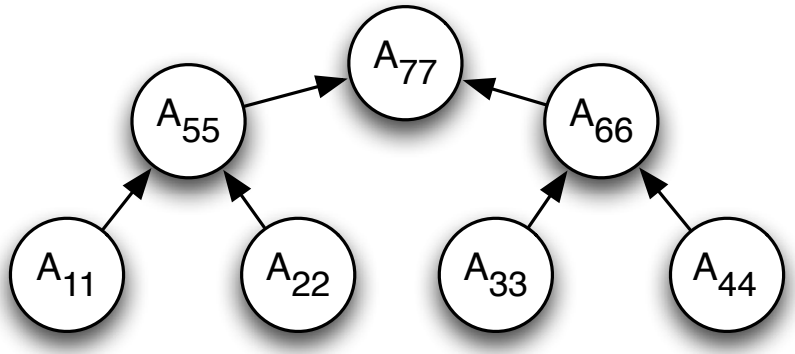
AXPY $\rightarrow$ on leaves
AXPY $\rightarrow$ on leaves
Preconditioning $\rightarrow$ cross. up + down

DOT $\rightarrow$ on leaves + reduction

AXPY-like $\rightarrow$ on leaves
2-norm $\rightarrow$ on leaves + reduction

Preconditioned Conjugate Gradient on OmpSs

Task dependency tree:



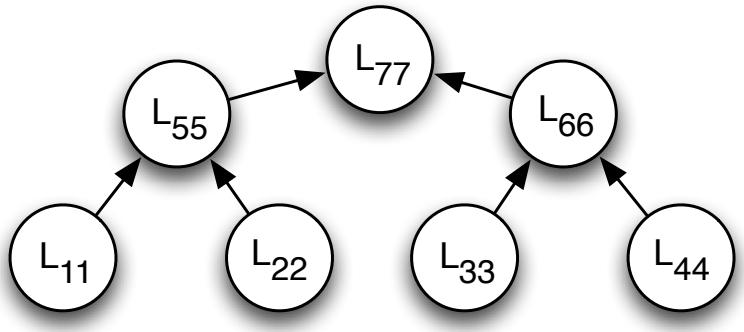
We define a structure called dag[][] to define the dependencies in OmpSs

Block		A_{11}	A_{22}	A_{33}	A_{44}	A_{55}	A_{66}	A_{77}
Task		T_0	T_1	T_2	T_3	T_4	T_5	T_6
Task id. j		0	1	2	3	4	5	6
Left descendant	`dag[0][`j`]`	0	1	2	3	0	2	4
Right descendant	`dag[1][`j`]`	0	1	2	3	1	3	5
Ancestor	`dag[2][`j`]`	4	4	5	5	6	6	6

The dag is created and initialized before the parallel computation begins

Preconditioned Conjugate Gradient on OmpSs

Task dependency tree:

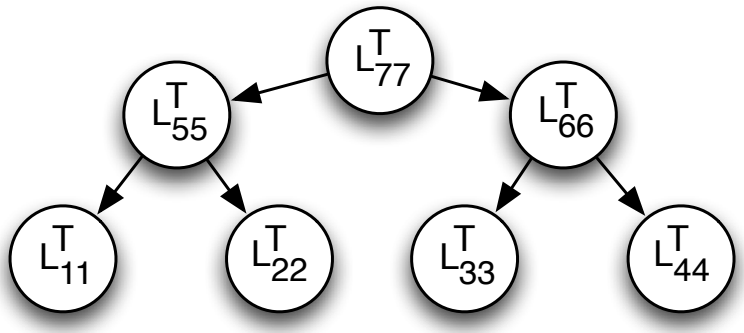


In the forward resolution, the tree is crossed from bottom to top:

```
for(i=0; i<dimL; i++){
  tsk_id=tsk_ord[i]; prio=tsk_prio[i];
  left=dag[0][tsk_id]; right=dag[1][tsk_id];
  #pragma omp task in(v_sentinel[left],v_sentinel[right]) \
                   out(v_sentinel[tsk_id]) priority(prio)
  ILUSolveUp(spFact, r, y, tsk_id,  v_sentinel, dag);
}
```

Preconditioned Conjugate Gradient on OmpSs

Task dependency tree:

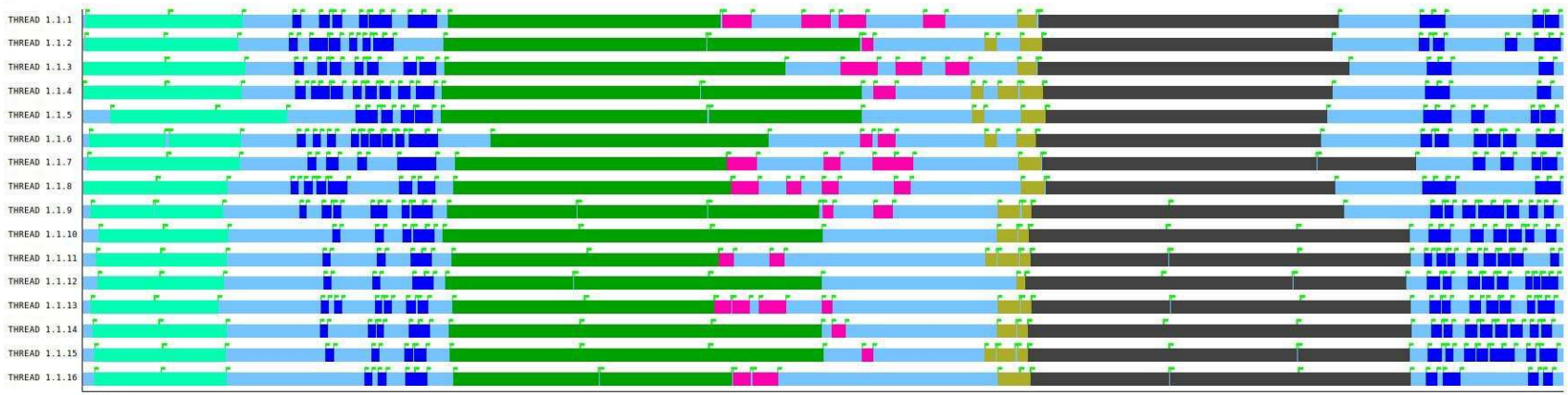


In the backward resolution, the tree is crossed from top to bottom:

```
for(i=0; i<dimL; i++){
  tsk_id=tsk_ord[i]; prio=tsk_prio[i];
  father=dag[2][tsk_id];
  #pragma omp task in(v_sentinel[father]) \
                   out(v_sentinel[tsk_id]) priority(prio)
  ILUSolveDown(spFact, y, z, tsk_id, v_sentinel, dag);
}
```

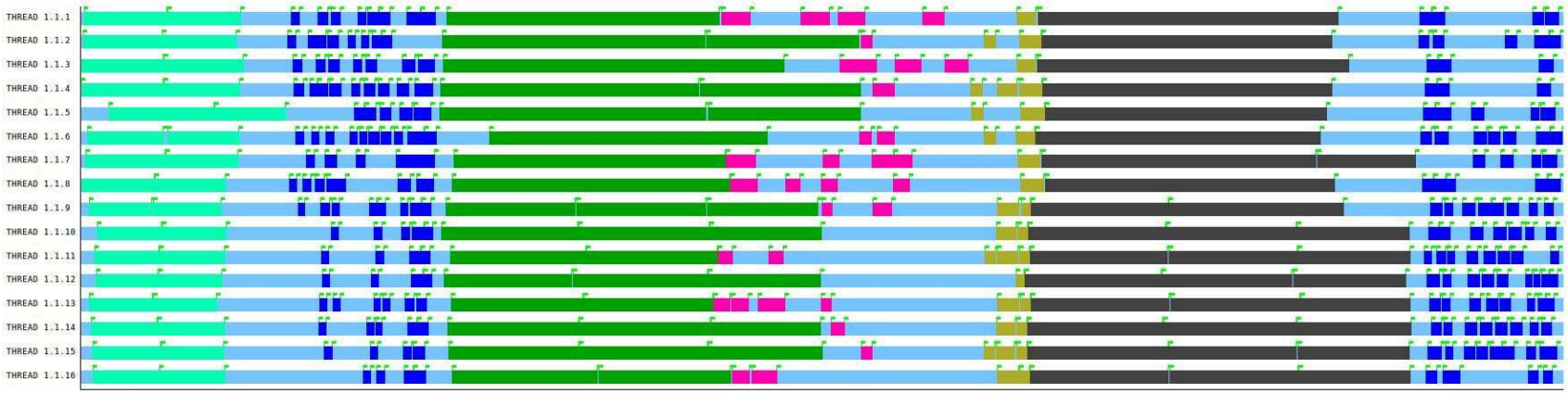
Preconditioned Conjugate Gradient on OmpSs

PARAVER trace of one iteration (32 leaves & 16 threads): many small tasks

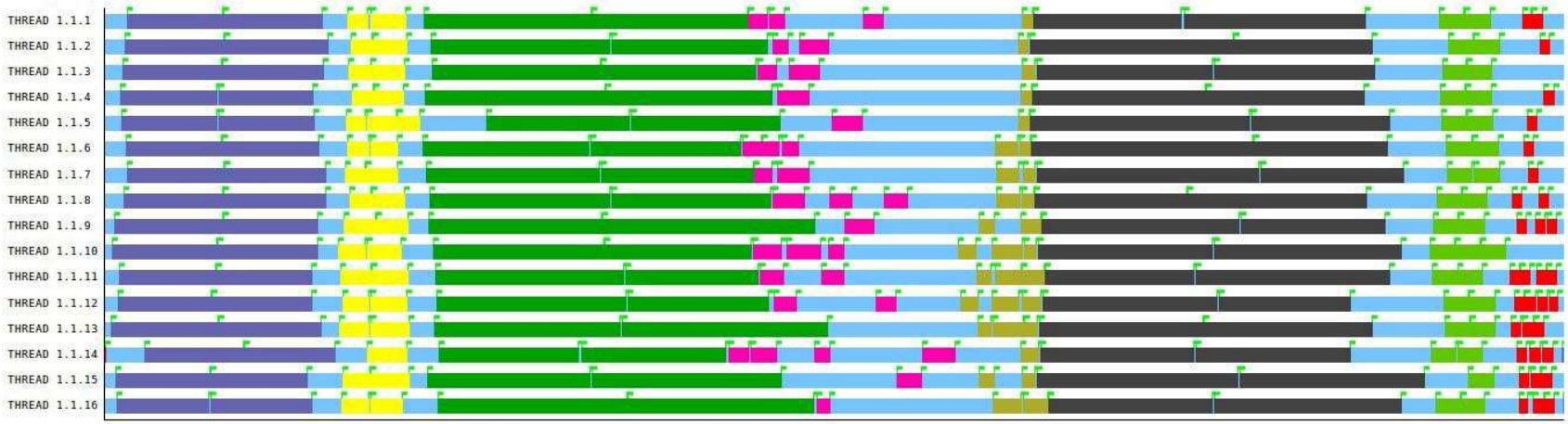


Preconditioned Conjugate Gradient on OmpSs

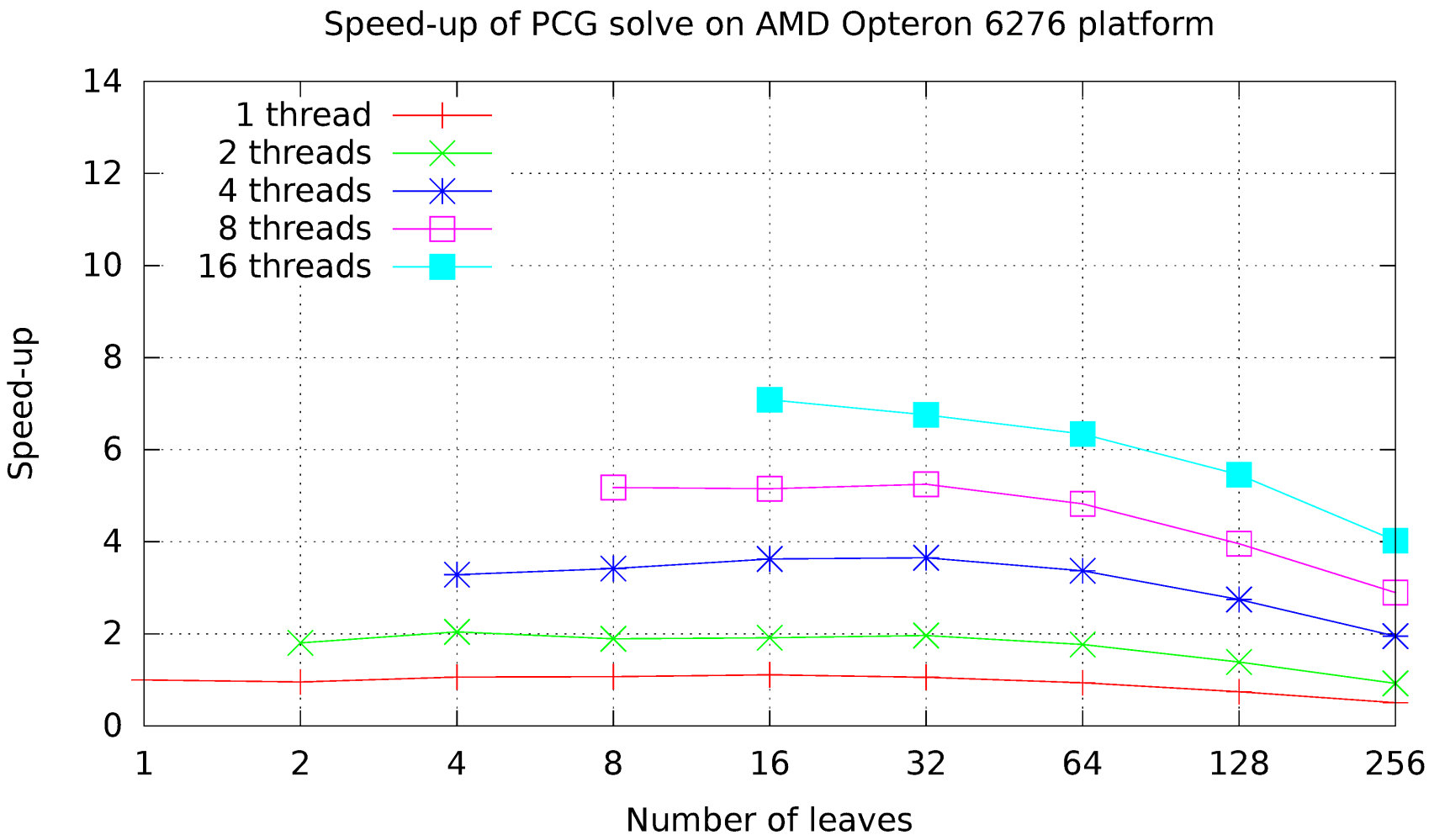
PARAVER trace of one iteration (32 leaves & 16 threads): many small tasks



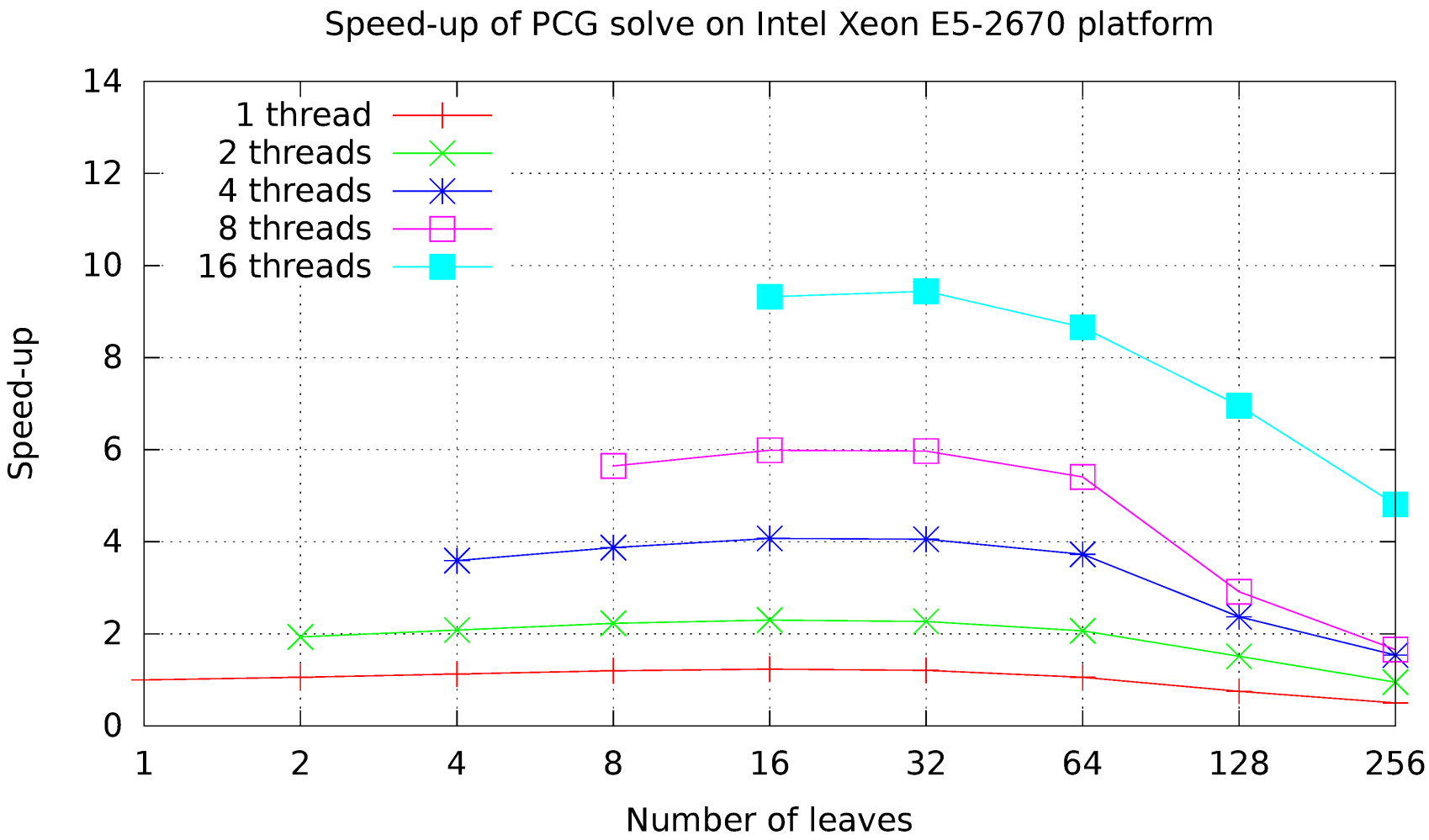
Merging small tasks:



OMPSS solver speed-up in AMD



OMPSS solver speed-up in INTEL

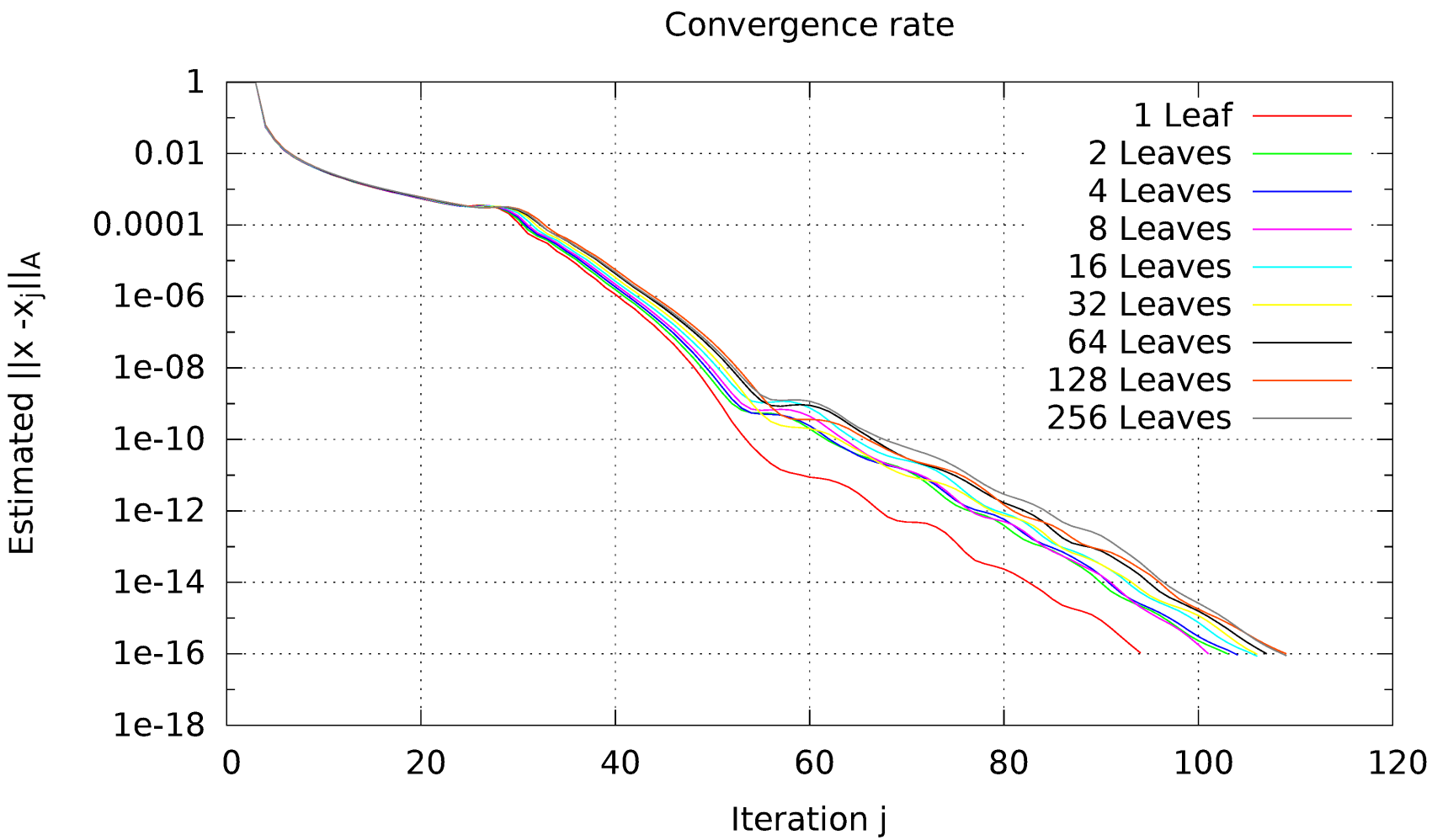


Parallel vs Sequential Solver

The conjugate gradient method minimizes at the jth step the energy norm of the error on the given j-dimensional Krylov subspace (Hestenes, 1952).

- We can estimate its value to analyze the behavior of each preconditioner.

Parallel vs Sequential Solver



Conclusions and Future work

The ILUPACK parallelization with OmpSs . . .

- Requires minor changes in the original code of ILUPACK
- Preserves the semantics of the method in ILUPACK
- Reduces the computation time by using the dynamic scheduling
- Improves the performances when # leaves are the twice # cores

Conclusions and Future work

The ILUPACK parallelization with OmpSs . . .

- Requires minor changes in the original code of ILUPACK
- Preserves the semantics of the method in ILUPACK
- Reduces the computation time by using the dynamic scheduling
- Improves the performances when # leaves are the twice # cores

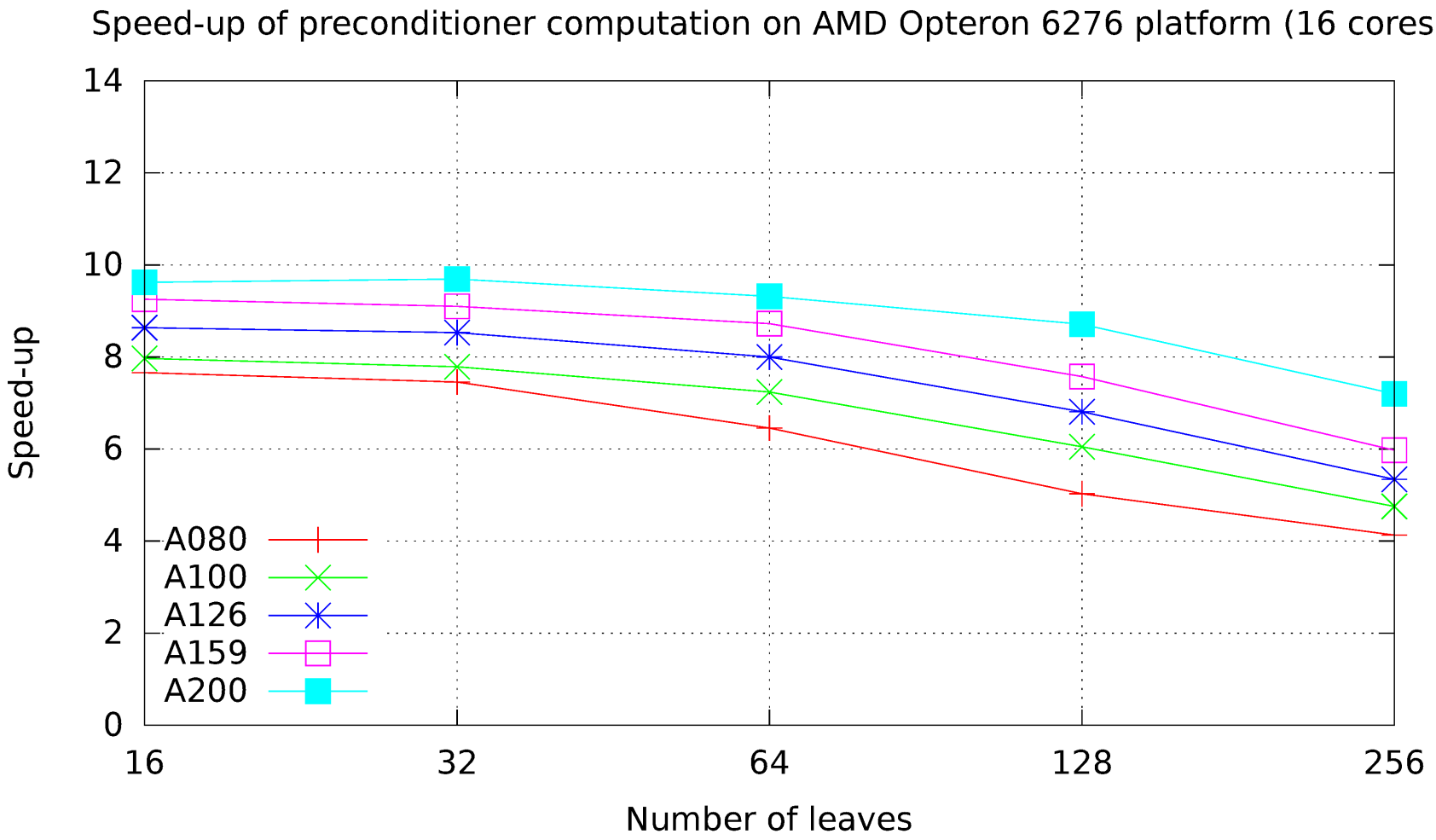
Future work . . .

- Investigate the effect of different options of the OmpSs runtime
  $\rightarrow$ Schedules that may favor data locality
  $\rightarrow$ Study how to improve the affinity in NUMA architectures
- Analyze the effect of power saving techniques on the performance
- Integrate energy saving mechanism into the OmpsS runtime scheduler
- Verify the distributed-memory and heterogeneous runtimes of OmpSs

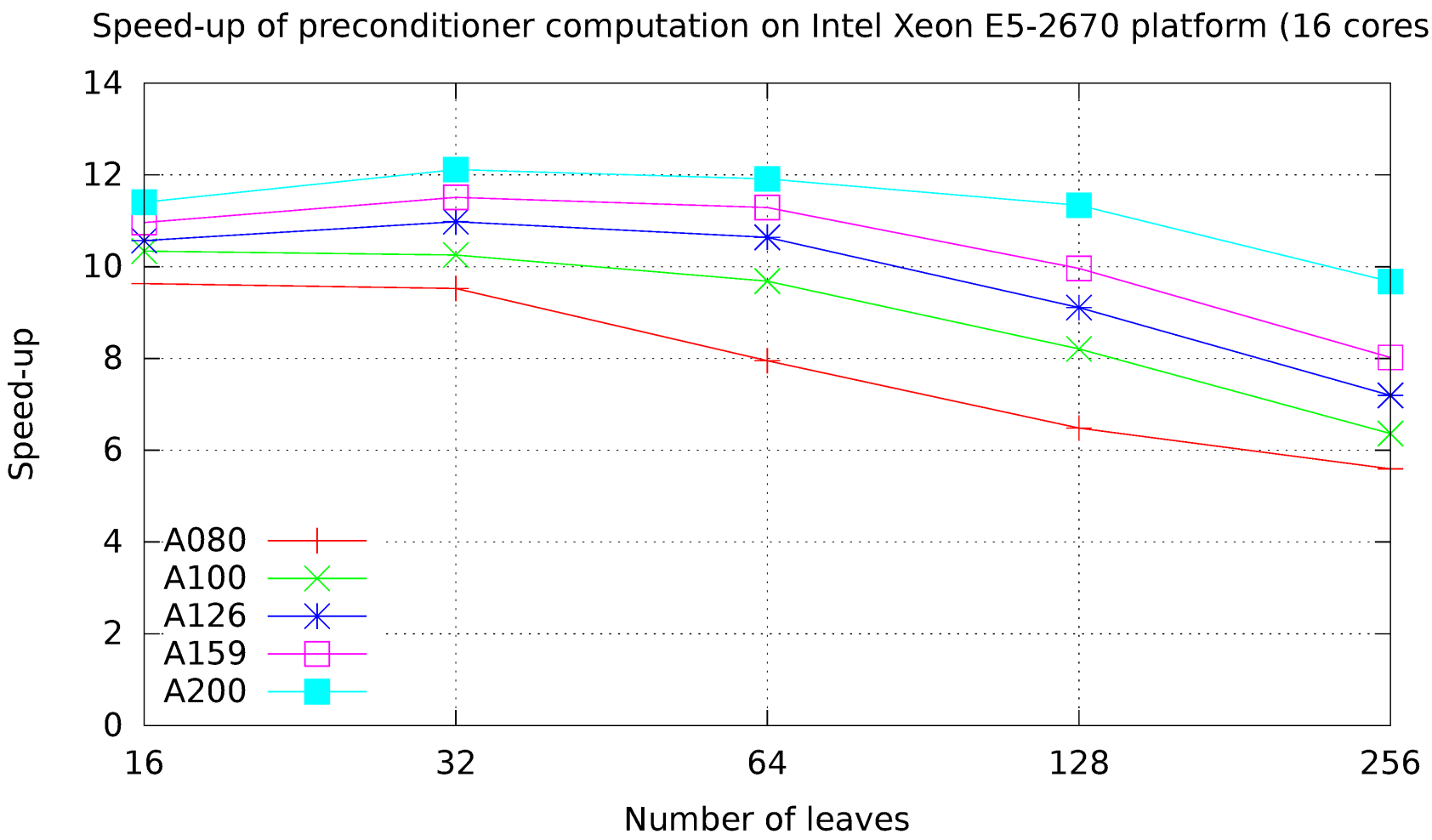
Thanks for your attention !

Questions ?

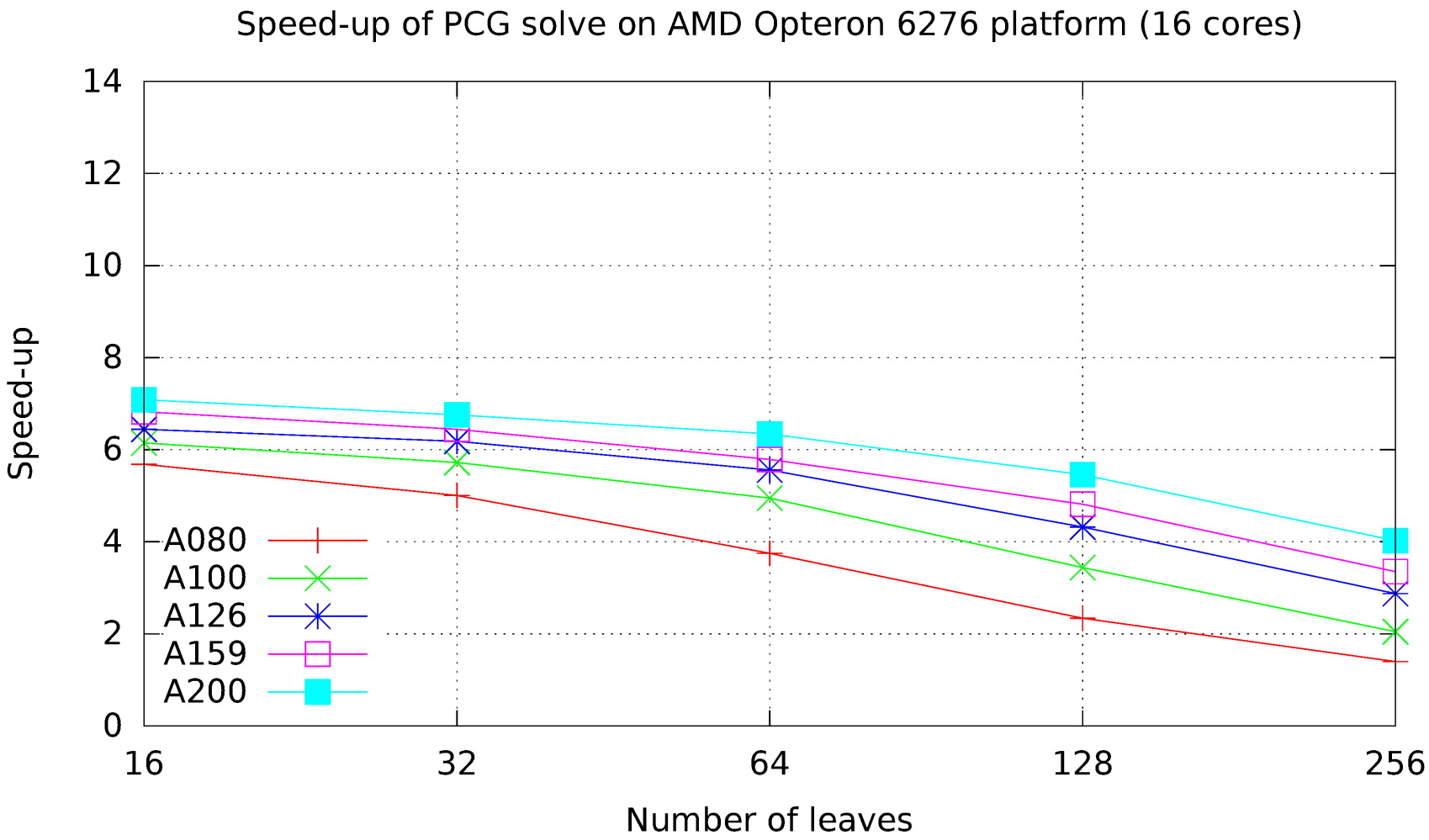
OMPSS preconditioner computation speed-up in AMD



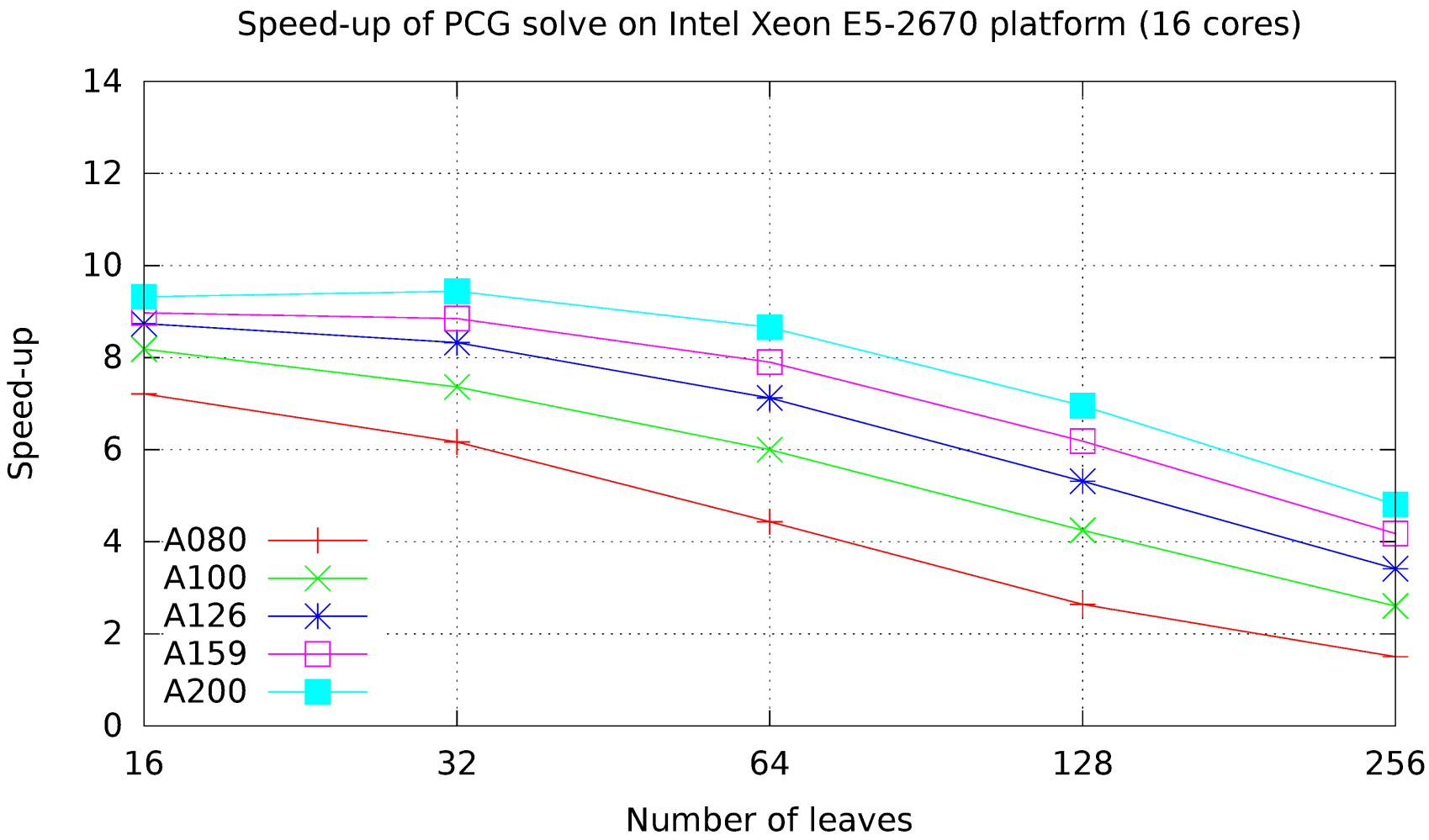
OMPSS preconditioner computation speed-up in INTEL



OMPSS solver speed-up in AMD



OMPSS solver speed-up in INTEL



Parallel Preconditioned Conjugate Gradient

- The data is distributed on the leaves of the elimination tree:
  - Consistently: the information is replicated:

$$x_c = \begin{pmatrix} x_1 \\ x_2 \\ \hline x_3 \end{pmatrix} \rightarrow \left(\frac{x_1}{x_3} \right), \left(\frac{x_2}{x_3} \right).$$

  - Unconsistently: some information isn't replicated:

$$b_u = \begin{pmatrix} b_1 \\ b_2 \\ \hline b_3 \end{pmatrix} \rightarrow \left(\frac{b_1}{b_3^1} \right), \left(\frac{b_2}{b_3^2} \right), b_3^1 + b_3^2 = b_3.$$

Parallel Preconditioned Conjugate Gradient

- The data is distributed on the leaves of the elimination tree:
  - Consistently: the information is replicated:

$$x_c = \begin{pmatrix} x_1 \\ x_2 \\ \hline x_3 \end{pmatrix} \rightarrow \left(\frac{x_1}{x_3} \right), \left(\frac{x_2}{x_3} \right).$$

  - Unconsistently: some information isn't replicated:

$$b_u = \begin{pmatrix} b_1 \\ b_2 \\ \hline b_3 \end{pmatrix} \rightarrow \left(\frac{b_1}{b_3^1} \right), \left(\frac{b_2}{b_3^2} \right), b_3^1 + b_3^2 = b_3.$$

Parallel Preconditioned Conjugate Gradient

- The data is distributed on the leaves of the elimination tree:
  - Consistently: the information is replicated:

$$x_c = \begin{pmatrix} x_1 \\ x_2 \\ \hline x_3 \end{pmatrix} \rightarrow \left(\frac{x_1}{x_3} \right), \left(\frac{x_2}{x_3} \right).$$

  - Unconsistently: some information isn't replicated:

$$b_u = \begin{pmatrix} b_1 \\ b_2 \\ \hline b_3 \end{pmatrix} \rightarrow \left(\frac{b_1}{b_3^1} \right), \left(\frac{b_2}{b_3^2} \right), b_3^1 + b_3^2 = b_3.$$

Parallel Preconditioned Conjugate Gradient

- The data is distributed on the leaves of the elimination tree:
  - Consistently: the information is replicated:

$$x_c = \left(\begin{array}{c} x_1 \\ x_2 \\ \hline x_3 \end{array}\right) \rightarrow \left(\frac{x_1}{x_3}\right), \left(\frac{x_2}{x_3}\right).$$

  - Unconsistently: some information isn't replicated:

$$b_u = \left(\begin{array}{c} b_1 \\ b_2 \\ \hline b_3 \end{array}\right) \rightarrow \left(\frac{b_1}{b_3^1}\right), \left(\frac{b_2}{b_3^2}\right), b_3^1 + b_3^2 = b_3.$$

$$A_u = \left(\begin{array}{cc|c} A_{11} & 0 & A_{31}^T \\ 0 & A_{22} & A_{32}^T \\ \hline A_{31} & A_{32} & A_{33} \end{array}\right) \rightarrow \left(\begin{array}{c|c} A_{11} & A_{31}^T \\ \hline A_{31} & A_{33}^1 \end{array}\right), \left(\begin{array}{c|c} A_{22} & A_{32}^T \\ \hline A_{32} & A_{33}^2 \end{array}\right), \quad A_{33}^1 + A_{33}^2 = A_{33}.$$

Parallel Preconditioned Conjugate Gradient

- The distribution defines how the computations are made:
  - The AXPY-like operations are made on similar operators.
  - The DOT-like operations are made on different operators:
  - The SPMV operations works as DOT-like operations:

Parallel Preconditioned Conjugate Gradient

- The distribution defines how the computations are made:
  - The AXPY-like operations are made on similar operators.
  - The DOT-like operations are made on different operators:
  - The SPMV operations works as DOT-like operations:

Parallel Preconditioned Conjugate Gradient

- The distribution defines how the computations are made:
  - The AXPY-like operations are made on similar operators.
  - The DOT-like operations are made on different operators:

$$b_u^T * x_c = \left(\begin{array}{c} b_1 \\ \hline b_2 \\ \hline b_3 \end{array}\right)^T * \left(\begin{array}{c} x_1 \\ \hline x_2 \\ \hline x_3 \end{array}\right)$$

$$b_u^T * x_c = \left(\frac{b_1}{b_3^1}\right)^T * \left(\frac{x_1}{x_3}\right) + \left(\frac{b_2}{b_3^2}\right)^T * \left(\frac{x_2}{x_3}\right)$$

$$b_u^T * x_c = val_1 + val_2$$

  - The SPMV operations works as DOT-like operations:

Parallel Preconditioned Conjugate Gradient

- The distribution defines how the computations are made:
  - The AXPY-like operations are made on similar operators.
  - The DOT-like operations are made on different operators:

$$b_u^T * x_c = \left(\begin{array}{c} b_1 \\ \hline b_2 \\ \hline b_3 \end{array}\right)^T * \left(\begin{array}{c} x_1 \\ \hline x_2 \\ \hline x_3 \end{array}\right) = val_1 + val_2 = val_u$$

  - The SPMV operations works as DOT-like operations:

Parallel Preconditioned Conjugate Gradient

- The distribution defines how the computations are made:
  - The AXPY-like operations are made on similar operators.
  - The DOT-like operations are made on different operators:

$$b_u^T * x_c = \left(\begin{array}{c} b_1 \\ \hline b_2 \\ \hline b_3 \end{array}\right)^T * \left(\begin{array}{c} x_1 \\ \hline x_2 \\ \hline x_3 \end{array}\right) = val_1 + val_2 = val_u$$

  - The SPMV operations works as DOT-like operations:

$$A_u * x_c = \left(\begin{array}{cc|c} A_{11} & 0 & A_{31}^T \\ 0 & A_{22} & A_{32}^T \\ \hline A_{31} & A_{32} & A_{33} \end{array}\right) * \left(\begin{array}{c} x_1 \\ \hline x_2 \\ \hline x_3 \end{array}\right)$$

$$A_u * x_c = \left(\begin{array}{c|c} A_{11} & A_{31}^T \\ \hline A_{31} & A_{33}^1 \end{array}\right) * \left(\begin{array}{c} x_1 \\ \hline x_3 \end{array}\right) \oplus \left(\begin{array}{c|c} A_{22} & A_{32}^T \\ \hline A_{32} & A_{33}^2 \end{array}\right) * \left(\begin{array}{c} x_2 \\ \hline x_3 \end{array}\right)$$

$$A_u * x_c = \left(\begin{array}{c} v_1 \\ \hline v_3^1 \end{array}\right) \oplus \left(\begin{array}{c} v_2 \\ \hline v_3^2 \end{array}\right) = v_u$$

Parallel Preconditioned Conjugate Gradient

- The distribution defines how the computations are made:
  - The AXPY-like operations are made on similar operators.
  - The DOT-like operations are made on different operators:

$$b_u^T * x_c = \left(\begin{array}{c} b_1 \\ b_2 \\ \hline b_3 \end{array}\right)^T * \left(\begin{array}{c} x_1 \\ x_2 \\ \hline x_3 \end{array}\right) = val_1 + val_2 = val_u$$

  - The SPMV operations works as DOT-like operations:

$$A_u * x_c = \left(\begin{array}{cc|c} A_{11} & 0 & A_{31}^T \\ 0 & A_{22} & A_{32}^T \\ \hline A_{31} & A_{32} & A_{33} \end{array}\right) * \left(\begin{array}{c} x_1 \\ x_2 \\ \hline x_3 \end{array}\right)$$

$$A_u * x_c = \left(\begin{array}{c} v_1 \\ 0 \\ \hline v_3^1 \end{array}\right) + \left(\begin{array}{c} 0 \\ v_2 \\ \hline v_3^2 \end{array}\right) = v_u$$

Parallel Preconditioned Conjugate Gradient

Initialize $k, x_0, r_0, z_0, d_0, \beta_0, \tau_0$
while $((k < k_{max})\&\&(\tau_k > \tau_{min}))$
$w_k := Ad_k$
$\gamma_k := d_k^T w_k$
$\rho_k := \beta_k / \gamma_k$
$x_{k+1} := x_k + \rho_k d_k$
$r_{k+1} := r_k - \rho_k w_k$
$z_{k+1} := M^{-1} r_{k+1}$
$\alpha_k := \beta_k$
$\beta_{k+1} := r_{k+1}^T z_{k+1}$
$\alpha_k := \beta_{k+1} / \alpha_k$
$d_{k+1} := z_{k+1} + \alpha_k d_k$
$\tau_{k+1} := ||r_{k+1}||_2$
$k := k + 1$
end while

At the beginning: A_u, x_c, b_u
Loop:
d_c, w_u
$\gamma_u \xrightarrow{reduction} \gamma_c$
ρ_c
x_c
r_u
$\xrightarrow{crossing} z_c$
α_c
$\beta_u \xrightarrow{reduction} \beta_c$
α_c

$\xrightarrow{crossing} r_c$, $\tau_u \xrightarrow{reduction} \tau_c$

Parallel Preconditioned Conjugate Gradient

Initialize $k, x, r, z, d, \beta, \tau$
while $((k < k_{max})\&\&(\tau > \tau_{min}))$
$w_u := A_u d_c$
$\gamma_c := d_c^T w_u$
$\rho_c := \beta_c / \gamma_c$
$x_c := x_c + \rho_c d_c$
$r_u := r_u - \rho_c w_u$
$\alpha_c := \beta_c$
$z_c := M^{-1} r_u$, $r_c \leftarrow r_u$
$\tau_c := sqrt(r_u^T r_c)$
$\beta_c := r_u^T z_c$
$\alpha_c := \beta_c / \alpha_c$
$d_c := z_c + \alpha_c d_c$
$k := k + 1$
end while

At the beginning: A_u, x_c, b_u
Loop:

$\gamma_u \xrightarrow{reduction} \gamma_c$

$\xrightarrow{crossing} z_c$, $\xrightarrow{crossing} r_c$
$\tau_u \xrightarrow{reduction} \tau_c$
$\beta_u \xrightarrow{reduction} \beta_c$