Parallelism in Linnea

Henrik Barthels, Paolo Bientinesi

HPAC High Performance and Automatic Computing | RWTH AACHEN UNIVERSITY

Introduction: Linnea

- How to compute the following expressions?

$$
\begin{aligned}
b &:= (X^T X)^{-1} X^T y \\
b &:= (X^T M^{-1} X)^{-1} X^T M^{-1} y \\
x &:= W(A^T (AWA^T)^{-1} b - c) \\
x &:= \left(A^{-T} B^T B A^{-1} + R^T [\Lambda(Rz)] R\right)^{-1} A^{-T} B^T B A^{-1} y
\end{aligned}
$$

- High-level languages are easy to use, but performance is usually suboptimal.
- BLAS and LAPACK can be fast, but require a lot of expertise.

BLAS [DDC+90], **LAPACK** [AB+99]

- $y \leftarrow Ax + y$
- $C \leftarrow AB + C$
- $B \leftarrow A^{-1}B$
- $\dots$

Introduction: Linnea

Input

$$z := (X^T S^{-1} X)^{-1} X^T S^{-1} y \qquad S \text{ is symmetric positive definite}$$

Output

$$
\begin{aligned}
L &:= \text{chol}(S) && \text{(potrf)} \\
U_1 &:= L^{-1} X && \text{(trsm)} \\
(Q, R) &:= \text{qr}(U_1) && \text{(geqrf)} \\
u_2 &:= L^{-1} y && \text{(trsv)} \\
u_3 &:= Q^T u_2 && \text{(ormqr)} \\
z &:= R^{-1} u_3 && \text{(trsv)}
\end{aligned}
$$

`https://github.com/HPAC/linnea`

Linear Algebra Knowledge

- Properties: `trmm` vs. `gemm`
- Inference of properties: $A \quad B \rightarrow AB$
- Simplifications: $A^T \rightarrow A$ if Symmetric(A)
- Rewriting expressions:

$$X := A^TA + A^TB + B^TA \quad \rightarrow \quad \begin{array}{l} Y := B + A/2 \\ X := A^TY + Y^TA \end{array}$$

- Common subexpressions:

$$X := AB^{-T}C + B^{-1}A^T \quad \rightarrow \quad \begin{array}{l} Z := AB^{-T} \\ X := ZC + Z^T \end{array}$$

- Matrix chains:

$$\begin{array}{ll} (AB)c & \mathcal{O}(n^3) \\ A(Bc) & \mathcal{O}(n^2) \end{array}$$

Problem 1: Generation of Parallel Code

Problem 2: Parallel Code Generation

Problem 1: Generation of Parallel Code

Problem 2: Parallel Code Generation

Generation of Parallel Code

How to make use of parallelism?

- Threaded kernels.
- Kernels in parallel.
- Both.

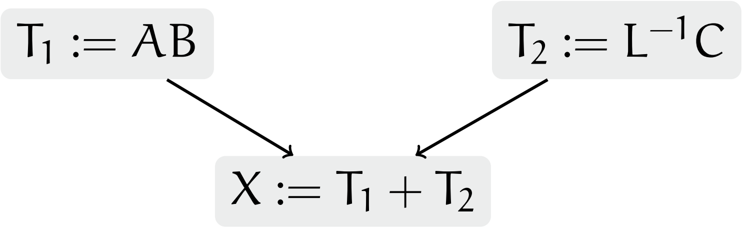
Generation of Parallel Code

Step 1: Parallelize a given sequence of kernels.

The Good

- Constructing dependency graph is easy.

$$
\begin{aligned}
T_1 &:= AB \\
T_2 &:= L^{-1}C \\
X &:= T_1 + T_2
\end{aligned}
$$



- If operand sizes are known, amount of work is known (#FLOPs).

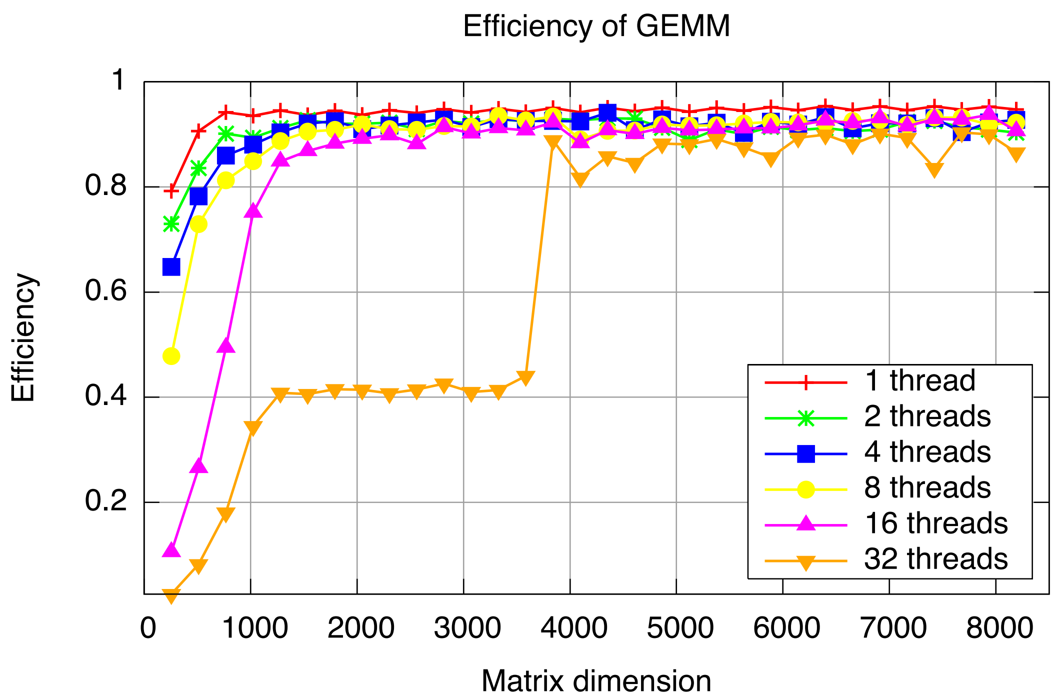
$$A \in \mathbb{R}^{m \times k}, B \in \mathbb{R}^{k \times n} \rightarrow AB \text{ requires } 2mnk \text{ FLOPs}$$

- BLAS and LAPACK are parallelized.

Generation of Parallel Code

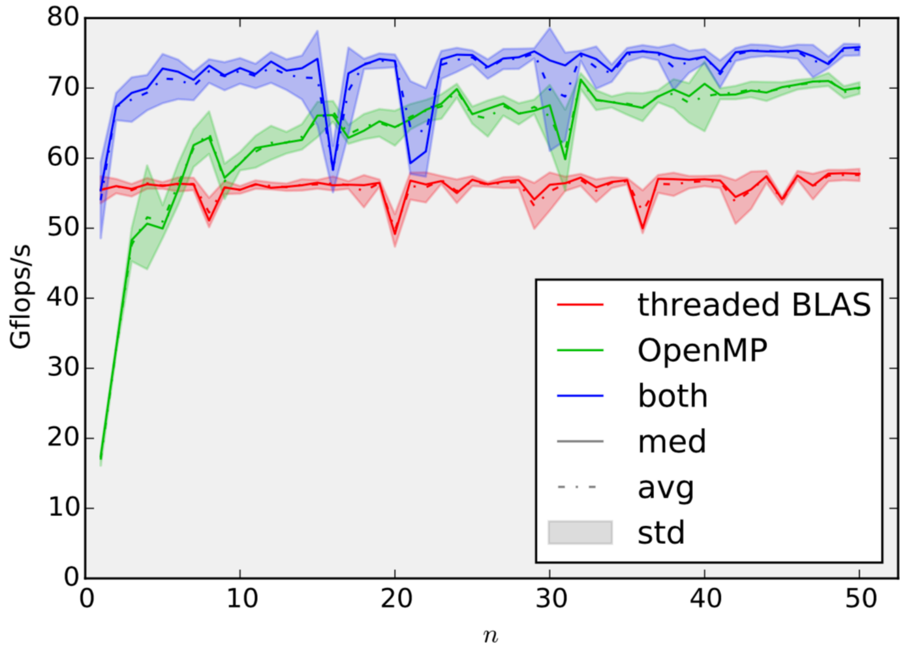
The Bad

- FLOP count is not a good prediction for execution time.
- Performance modeling is a hard problem [PB12].
  - Performance is not composable.
  - Efficiency decreases with number of threads.



The Bad

- FLOP count is not a good prediction for execution time.
- Performance modeling is a hard problem [PB12].
  - Performance is not composable.
  - Efficiency decreases with number of threads.
  - “Overbooking”: Parallelizing a sequence of n LU’s [PB15].



Generation of Parallel Code

Existing Tools

PaRSEC, OmpSs, StarPU, SuperGlue,…

- Built for large dependency graphs/large number of tasks.
- Only one thread per task.
- Do not consider cost (except StarPU [ATN09]).

We have:

- Small number of tasks.
- Multiple threads per task.
- Cost is (roughly) known.

Generation of Parallel Code

Step 2: Generate sequence that parallelizes well.

A good sequence of kernels for sequential execution may not be good for parallel execution.

Example: $X := ABCD$

$A, B, C \in \mathbb{R}^{n \times n}$

$D \in \mathbb{R}^{n \times m}$

$m < n$

$A(B(CD))$

- min #FLOPs
- dependencies
- threaded kernels still possible

$(AB)(CD)$

- more FLOPs
- fewer dependencies

Existing work: Matrix Chain Products on Parallel Systems [LKHL03]

Problem 1: Generation of Parallel Code

Problem 2: Parallel Code Generation

Parallel Code Generation

Motivation

- For sufficiently large matrices and/or enough runs of the program, generation time will be amortized.
- What about small computations?
- What about computations in interactive environment such as Matlab?

Parallel Code Generation

$$b := X^T M^{-1} X y$$

Parallel Code Generation

$$\begin{tikzcd}
& b := X^T M^{-1} X y \arrow[dl, "L := \mathsf{chol}(M)"'] \arrow[dr] & \\
b := X^T (L^T L)^{-1} X y & & \dots
\end{tikzcd}$$

Parallel Code Generation

$$b := X^T M^{-1} X y$$

$L := \text{chol}(M)$

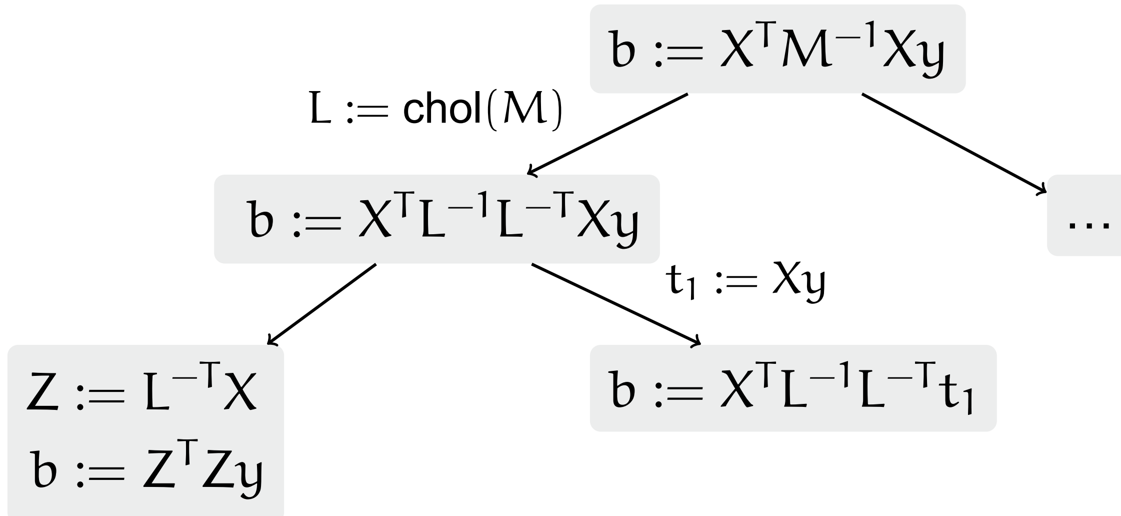
$$b := X^T L^{-1} L^{-T} X y$$

$$\ldots$$

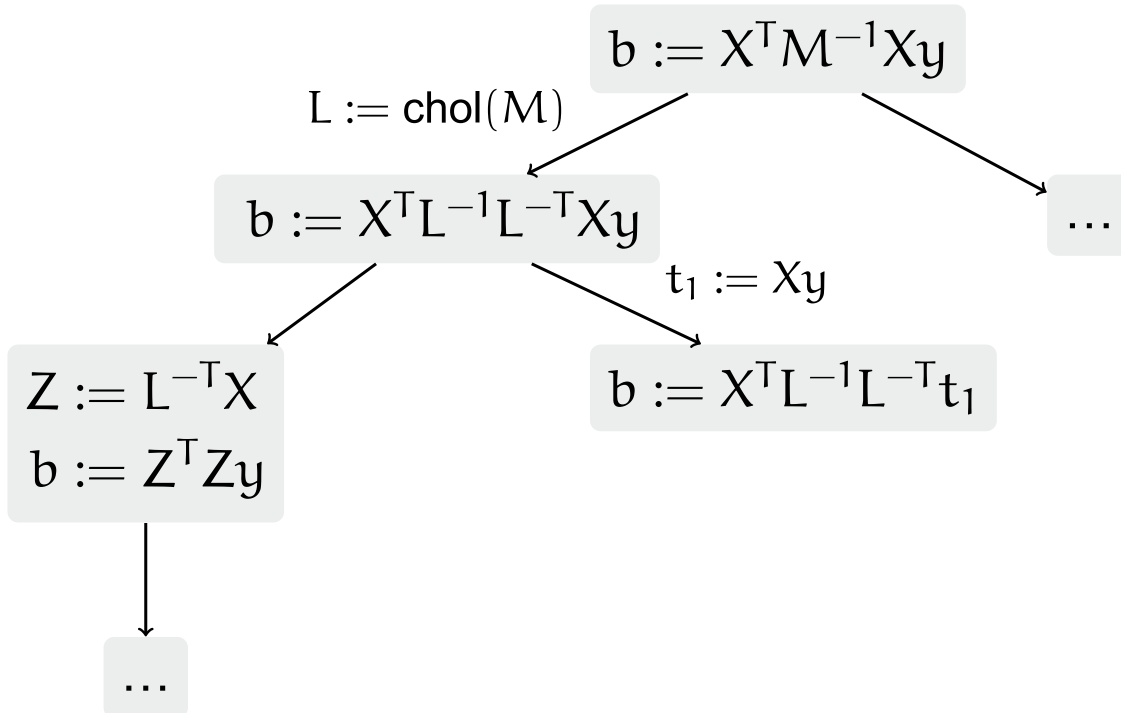
Parallel Code Generation

$$
\begin{tikzcd}
 & b := X^T M^{-1} X y \arrow[dl, "L := \text{chol}(M)"'] \arrow[dr] & \\
b := X^T L^{-1} L^{-T} X y \arrow[d] & & \dots \\
\begin{array}{l} Z := L^{-T} X \\ b := Z^T Z y \end{array} & &
\end{tikzcd}
$$

Parallel Code Generation



Parallel Code Generation



Parallel Code Generation

Reducing Redundancy

$$\begin{tikzcd}
 & X := ABC + CDE \arrow[dl, "T_1 := AB"'] \arrow[dr, "T_3 := BC"] & \\
X := T_1C + CDE \arrow[d, "T_2 := T_1C"'] & & X := AT_3 + CDE \arrow[d, "T_4 := AT_3"] \\
X := T_2 + CDE & & X := T_4 + CDE
\end{tikzcd}$$

Parallel Code Generation

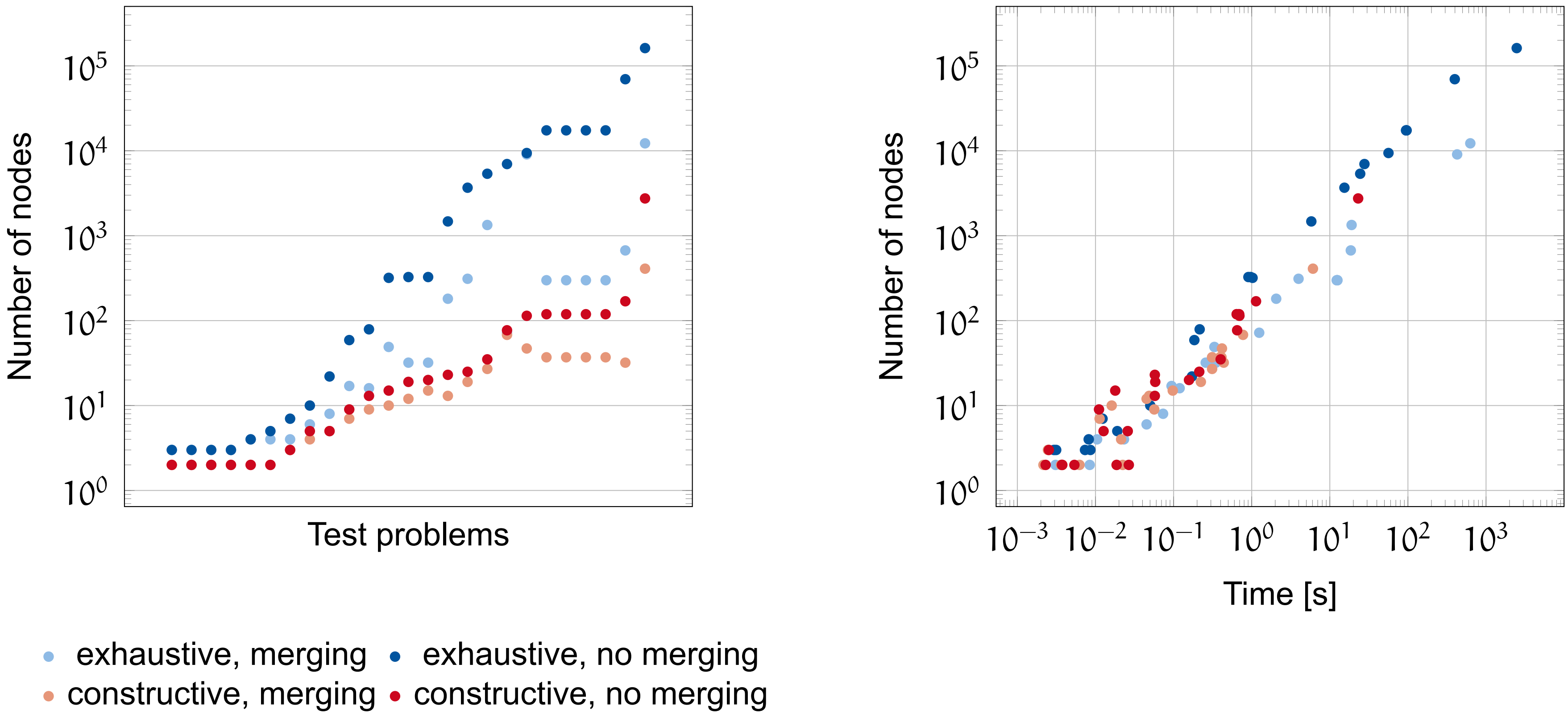
Reducing Redundancy

$$\begin{tikzcd}[column sep=small]
& X := ABC + CDE \arrow[dl, "T_1 := AB"'] \arrow[dr, "T_3 := BC"] & \\
X := T_1C + CDE \arrow[dr, "T_2 := T_1C"'] & & X := AT_3 + CDE \arrow[dl, "T_2 := AT_3"] \\
& X := T_2 + CDE &
\end{tikzcd}$$

Table of expressions and intermediate operands:

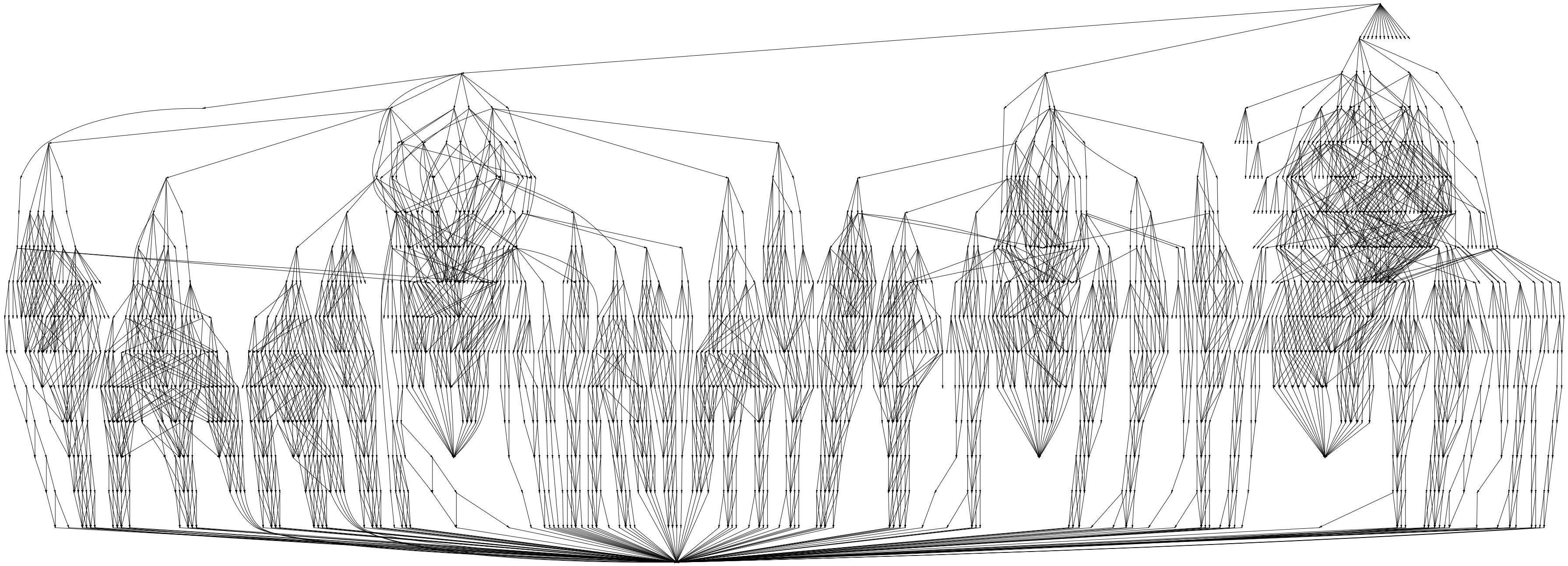
tmp	expr
T_1	AB
T_2	ABC
T_3	BC

Reducing Redundancy



Parallel Code Generation

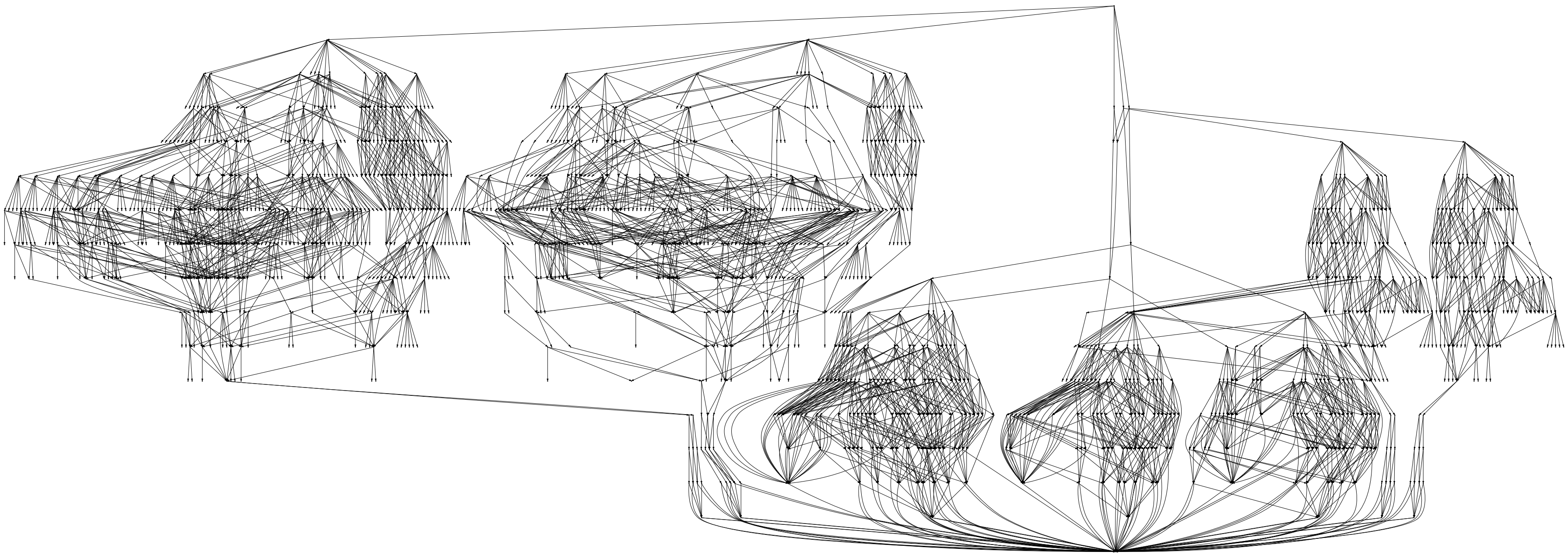
$$z := (X^T S^{-1} X)^{-1} X^T S^{-1} y \qquad S \text{ is symmetric positive definite}$$



Parallel Code Generation

$$x := W(A^T(AWA^T)^{-1}b - c)$$

W is diagonal, diagonal elements are positive



Parallel Code Generation

The Good

- Finding redundant nodes can be done efficiently using hash table.

The Bad

- Access to table of expressions & intermediates has to be protected.
- Merging nodes requires synchronization.
- Graph is not uniform at all.
- Graph is initially not known.

Possible solutions:

- Tradeoff between merging and redundancy?
- Parallelize computations on nodes?
- Nodes as tasks?

References

[AB+99] Edward Anderson, Zhaojun Bai, et al. *LAPACK Users' guide*, volume 9. SIAM, 1999.

[ATN09] Cédric Augonnet, Samuel Thibault, and Raymond Namyst. Automatic Calibration of Performance Models on Heterogeneous Multicore Architectures. *Euro-Par Workshops*, 6043(Chapter 9):56–65, 2009.

[DDC+90] Jack J. Dongarra, Jeremy Du Croz, et al. A set of Level 3 Basic Linear Algebra Subprograms. *ACM TOMS*, 16(1):1–17, 1990.

[LKHL03] Heejo Lee, Jong Kim, Sung Je Hong, and Sunggu Lee. Processor Allocation and Task Scheduling of Matrix Chain Products on Parallel Systems. *Parallel and Distributed Systems, IEEE Transactions on*, 14(4):394–407, 2003.

[PB12] Elmar Peise and Paolo Bientinesi. Performance Modeling for Dense Linear Algebra. In *Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis*

References

(PMBS12), SCC ’12, pages 406–416, Washington, DC, USA, November 2012. IEEE Computer Society.

[PB15] Elmar Peise and Paolo Bientinesi. The ELAPS Framework - Experimental Linear Algebra Performance Studies. *CoRR*, cs.PF, 2015.