Hybrid Scheduling for already Optimized Dense Matrix Factorization

Simplice Donfack, Laura Grigori
INRIA Saclay

Bill Gropp, Vivek Kale
UIUC

Plan

- Brief introduction of communication avoiding methods
- Hybrid scheduling for already optimized dense linear algebra (communication avoiding)
- Experiments on a 48 cores AMD Opteron machine
- Conclusions and future work

Motivation for Communication Avoiding Algorithms

- Time_per_flop << 1/ bandwidth << latency
  - Gaps growing exponentially with time

Annual improvements			
Time/flop		Bandwidth	Latency
59%	Network	26%	**15%**
	DRAM	23%	**5%**

- Communication avoiding algorithmic design: the communication minimization becomes part of the numerical algorithm design (in collaboration with J. Demmel)
- Better performance, less energy consumption

Algorithms and lower bounds on communication

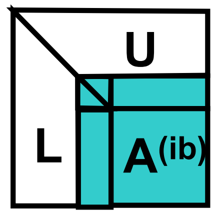
- Goals for CA algorithms:
  - Minimize #words_moved = Ω (#flops/ $M^{1/2}$) = Ω (n^2 / $P^{1/2}$)
  - Minimize #messages = Ω (#flops/ $M^{3/2}$) = Ω ($P^{1/2}$)
  - Minimize over multiple levels of memory/parallelism
  - Allow redundant computations (preferably as a low order term)
- LAPACK and ScaLAPACK
  - mostly suboptimal (newer version starts implementing CA algorithms)
- Recursive cache oblivious algorithms
  - Minimize bandwidth, not latency, sometimes more flops (3x for QR)
- CA algorithms
  - Communication optimal for dense algorithms and some sparse algorithms

LU factorization (as in ScaLAPACK pdgetrf)

LU factorization on a $P = P_r \times P_c$ grid of processors

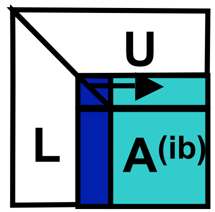
For i = 1 to n-1 step b

$A^{(ib)} = A(ib:n, ib:n)$



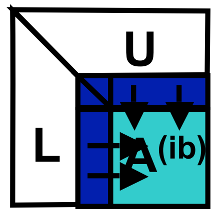
(1) Compute panel factorization (pdgetf2) $O(n \log_2 P_r)$

- find pivot in each column, swap rows



(2) Apply all row permutations (pdlaswp) $O(n/b(\log_2 P_c + \log_2 P_r))$

- broadcast pivot information along the rows
- swap rows at left and right

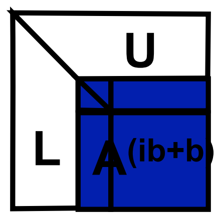


(3) Compute block row of U (pdtrsm) $O(n/b \log_2 P_c)$

- broadcast right diagonal block of L of current panel

(4) Update trailing matrix (pdgemm) $O(n/b(\log_2 P_c + \log_2 P_r))$

- broadcast right block column of L
- broadcast down block row of U

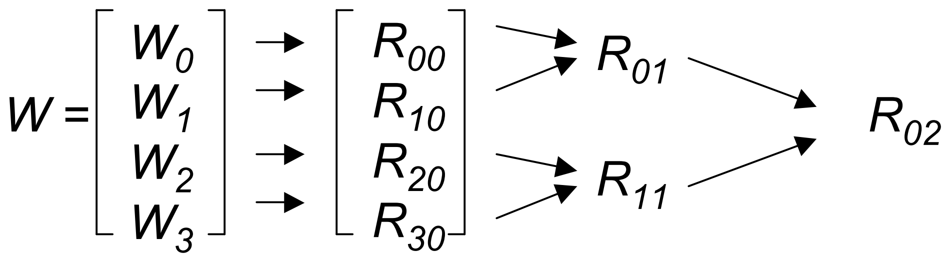


TSQR: QR factorization of a tall skinny matrix using Householder transformations

- QR decomposition of m x b matrix W, m >> b
  - P processors, block row layout

- Usual Parallel Algorithm
  - Compute Householder vector for each column
  - Number of messages ∝ b log P

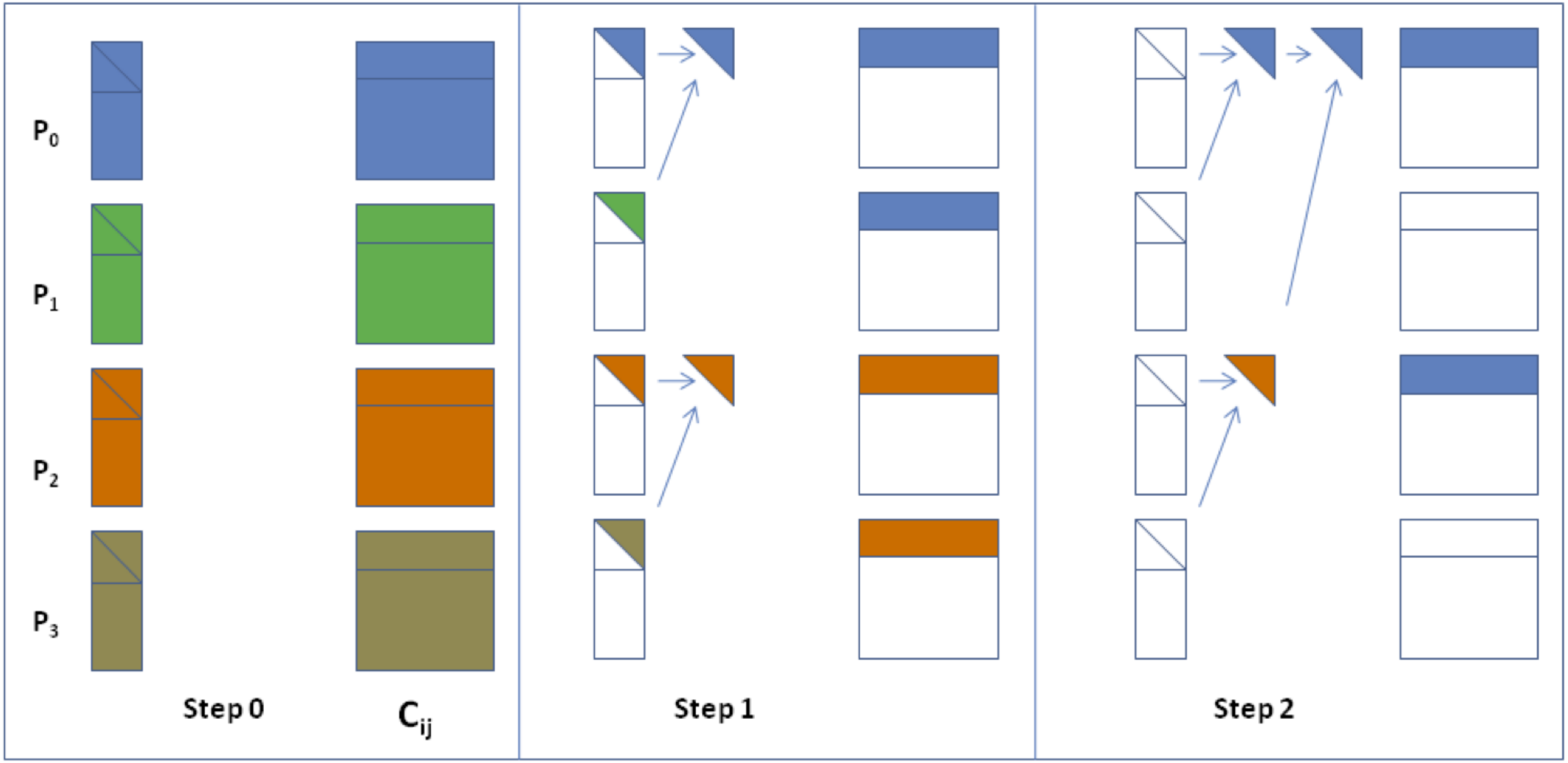
- Communication Avoiding Algorithm
  - Reduction operation, with QR as operator
  - Number of messages ∝ log P



J. Demmel, LG, M. Hoemmen, J. Langou '08

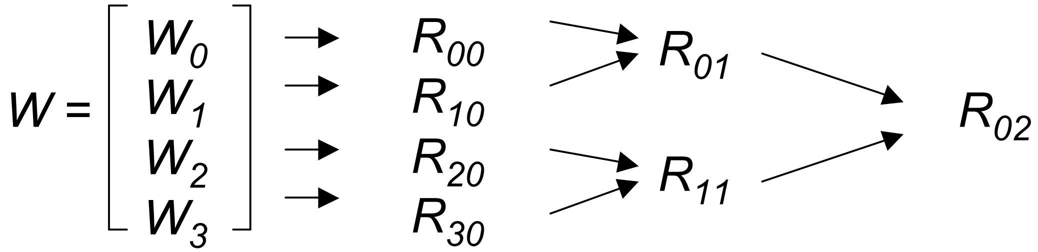
CAQR for general matrices

- Use TSQR for panel factorizations
- Update the trailing matrix - triggered by the reduction tree used for the panel factorization

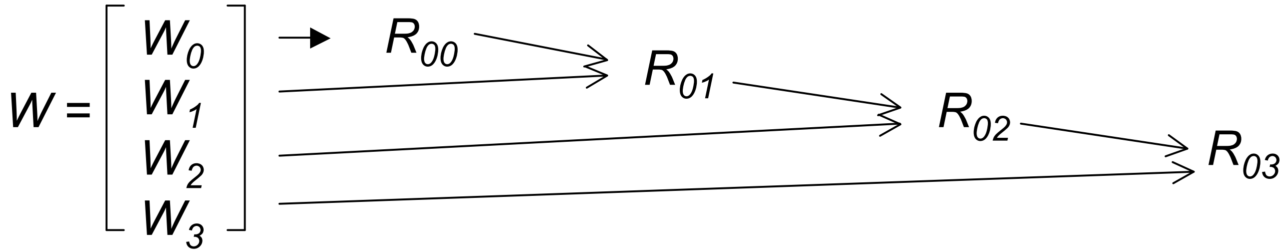


Flexibility of CAQR factorization

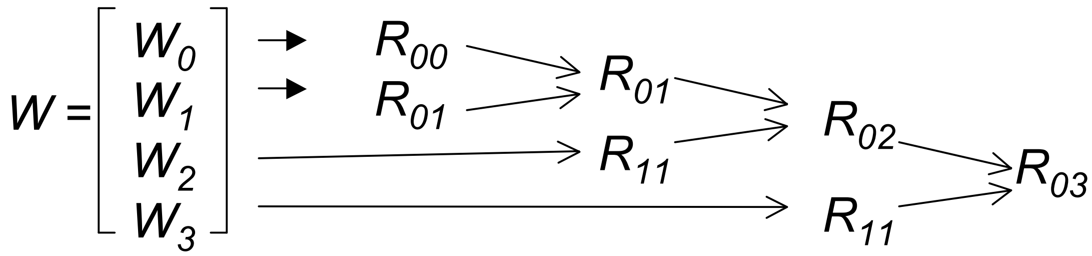
Parallel:



Sequential:



Dual Core:



Reduction tree will depend on the underlying architecture, could be chosen dynamically

Factorizations that require pivoting

- Using the idea from CAQR leads to unstable factorizations
- Requires new tournament pivoting scheme (LU, RRQR)
- Consider a block algorithm that factors an n-by-n matrix A.

$$
A = \begin{array}{c l}
\begin{array}{cc} b & n-b \end{array} & \\
\begin{pmatrix} \bar{A}_{11} & \bar{A}_{21} \\ A_{21} & A_{22} \end{pmatrix} & \begin{array}{l} \} \; b \\ \} \; n-b \end{array}
\end{array}
$$

, where

$$
W = \begin{pmatrix} A_{11} \\ A_{21} \end{pmatrix}
$$

- At each iteration
  - Preprocess W to find at low communication cost good pivots for the LU factorization of W.
  - Permute the pivots to top.
  - Compute LU with no pivoting of W, update trailing matrix.

$$
PA = \begin{pmatrix} L_{11} & \\ L_{21} & I_{n-b} \end{pmatrix} \begin{pmatrix} I_b & \\ & A_{22} - L_{21}U_{12} \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} \\ & I_{n-b} \end{pmatrix}
$$

Tournament pivoting for a tall skinny matrix

P$_0$

$$W_0 = \begin{pmatrix}2&4\\0&1\\2&0\\1&2\end{pmatrix} = \Pi_0 L_0 U_0 \qquad \Pi_0^T W_0 = \begin{pmatrix}2&4\\2&0\end{pmatrix}$$

$$\overline{W}_0 = \begin{pmatrix}2&4\\2&0\\4&1\\2&0\end{pmatrix} = \overline{\Pi}_0 \overline{L}_0 \overline{U}_0 \qquad \overline{\Pi}_0^T \overline{W}_0 = \begin{pmatrix}4&1\\2&4\end{pmatrix}$$

$$\underline{W}_0 = \begin{pmatrix}4&1\\2&4\\4&2\\1&4\end{pmatrix} = \underline{\Pi}_0 \underline{L}_0 \underline{U}_0 \qquad \underline{\Pi}_0^T \underline{W}_0 = \begin{pmatrix}4&1\\1&4\end{pmatrix}$$

Good pivots for factorizing W

P$_1$

$$W_1 = \begin{pmatrix}2&0\\0&0\\4&1\\1&0\end{pmatrix} = \Pi_1 L_1 U_1 \qquad \Pi_1^T W_1 = \begin{pmatrix}4&1\\2&0\end{pmatrix}$$

P$_2$

$$W_2 = \begin{pmatrix}0&1\\1&4\\0&0\\0&2\end{pmatrix} = \Pi_2 L_2 U_2 \qquad \Pi_2^T W_2 = \begin{pmatrix}1&4\\0&2\end{pmatrix}$$

$$\overline{W}_2 = \begin{pmatrix}1&4\\0&2\\4&2\\0&2\end{pmatrix} = \overline{\Pi}_2 \overline{L}_2 \overline{U}_2 \qquad \overline{\Pi}_2^T \overline{W}_2 = \begin{pmatrix}4&2\\1&4\end{pmatrix}$$

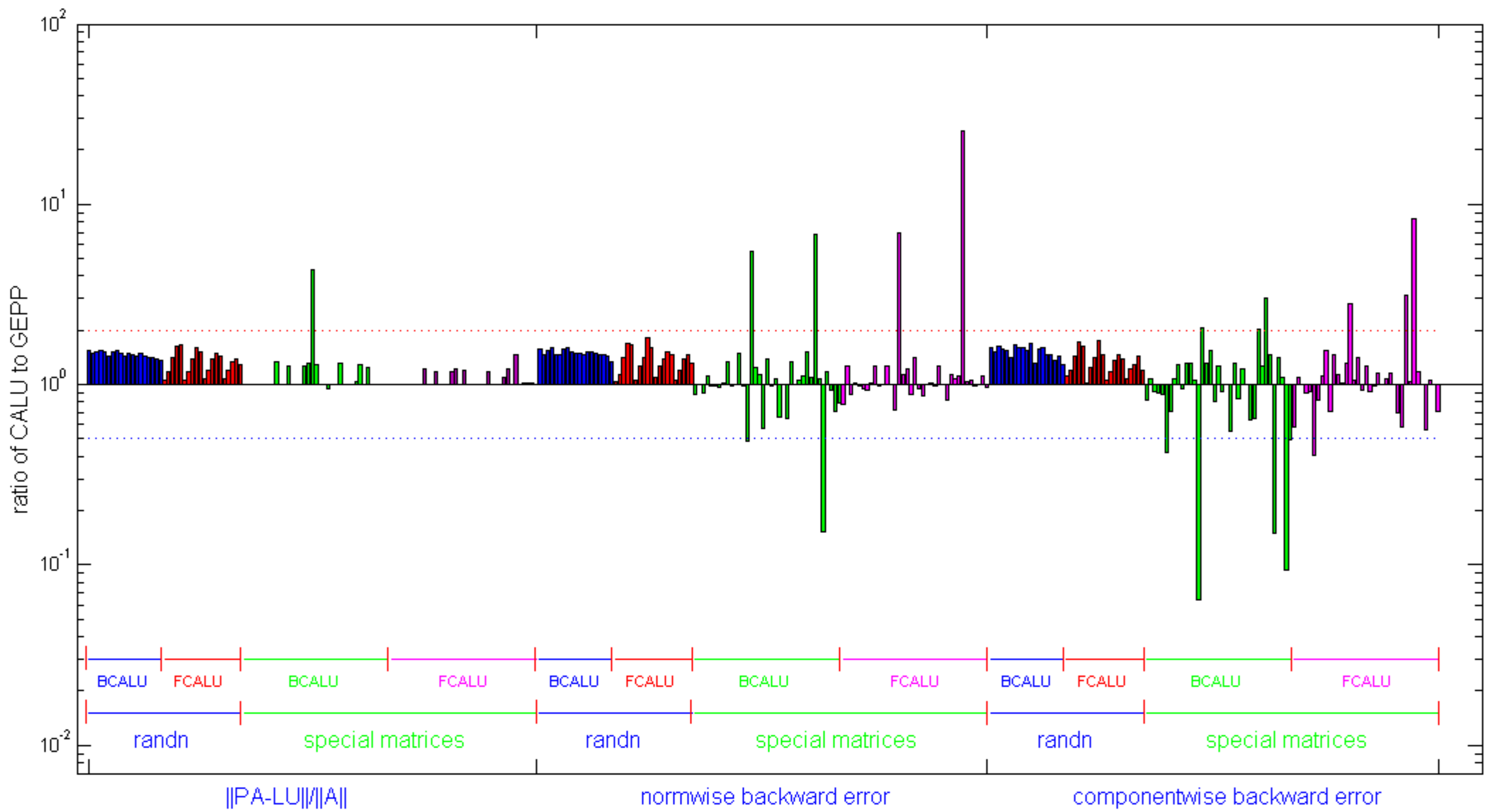
P$_3$

$$W_3 = \begin{pmatrix}2&1\\0&2\\1&0\\4&2\end{pmatrix} = \Pi_3 L_3 U_3 \qquad \Pi_3^T W_3 = \begin{pmatrix}4&2\\0&2\end{pmatrix}$$

time

Stability of CALU (experimental results)

- Results show ||PA-LU||/||A||, normwise and componentwise backward errors, for random matrices and special ones
  - See [LG, Demmel, Xiang, 2010] for details
  - BCALU denotes binary tree based CALU and FCALU denotes flat tree based CALU



CALU_PRRP: CALU with panel rank revealing pivoting

- Tournament pivoting uses strong RRQR at each node of the reduction tree
- Worst case analysis of growth factor
  - matrix of size m-by-n
  - reduction tree of height H=log(P).

CALU		GEPP	CALU_PRRP
Upper bound $2^{n(H+1)-1}$	Attained 2^{n-1}	Upper bound 2^{n-1}	Upper bound $(1+2b)^{(n/b)\log(P)}$

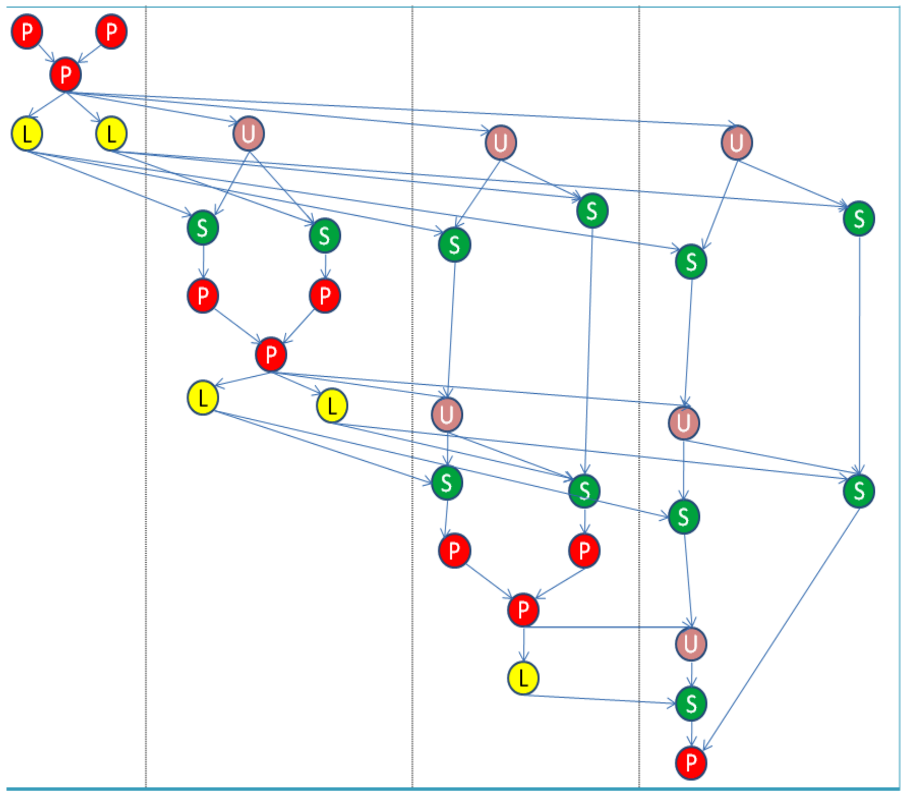
Better stability →

- CALU_PRRP stable for pathological cases and matrices from solving Volterra integral equation (Foster).

A. Khabou, LG, J. Demmel, M. Gu

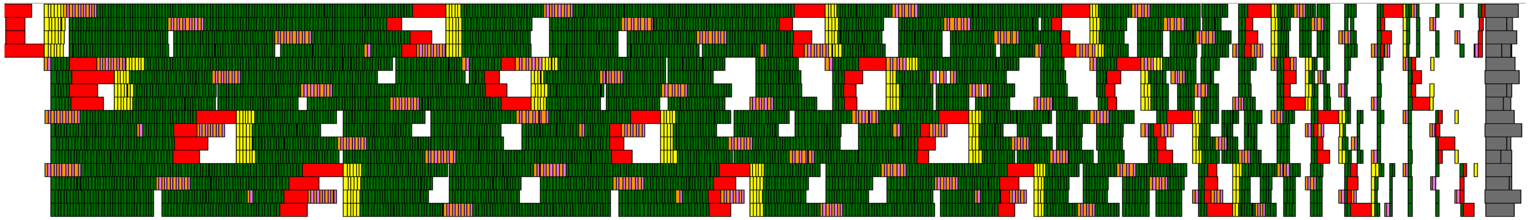
CALU and its task dependency graph

- The matrix is partitioned in blocks of size T x b.
- The computation of each block is associated with a task.
- The task dependency graph (DAG) can be executed using any scheduling strategy.



Scheduling CALU's Task Dependency Graph

- Static scheduling
  - \+ Good locality of data   - Ignores OS jitter

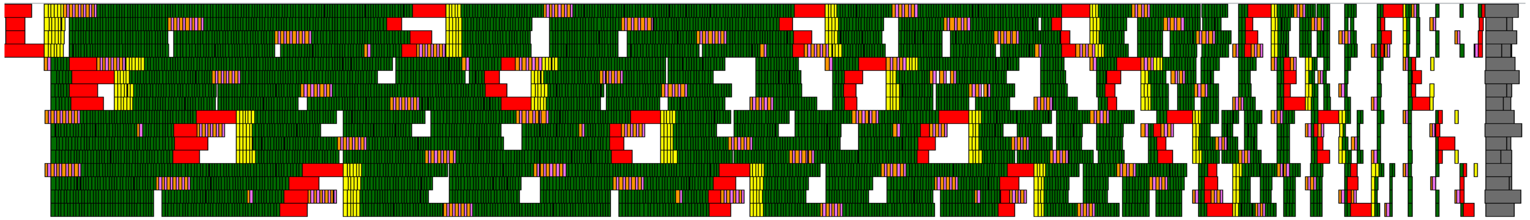


Scheduling CALU's Task Dependency Graph

- **Static scheduling**

  \+ Good locality of data

  \- Ignores OS jitter

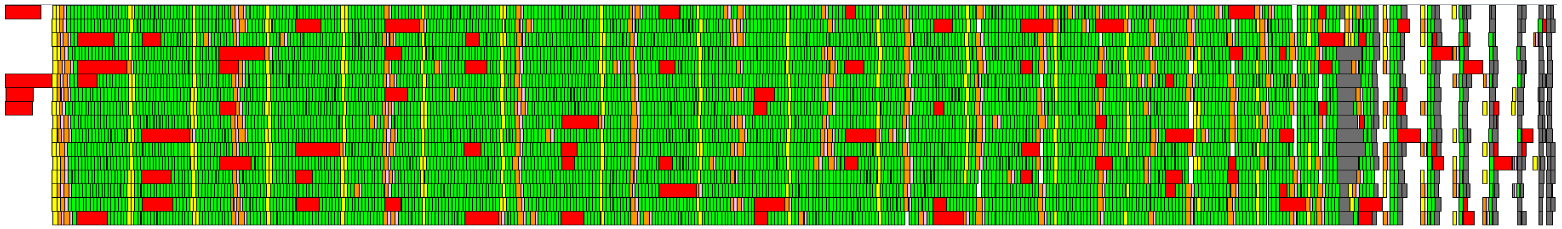


- **Dynamic scheduling**

  \+ Keeps cores busy

  \- Poor usage of data locality

  \- Can lead to large overhead



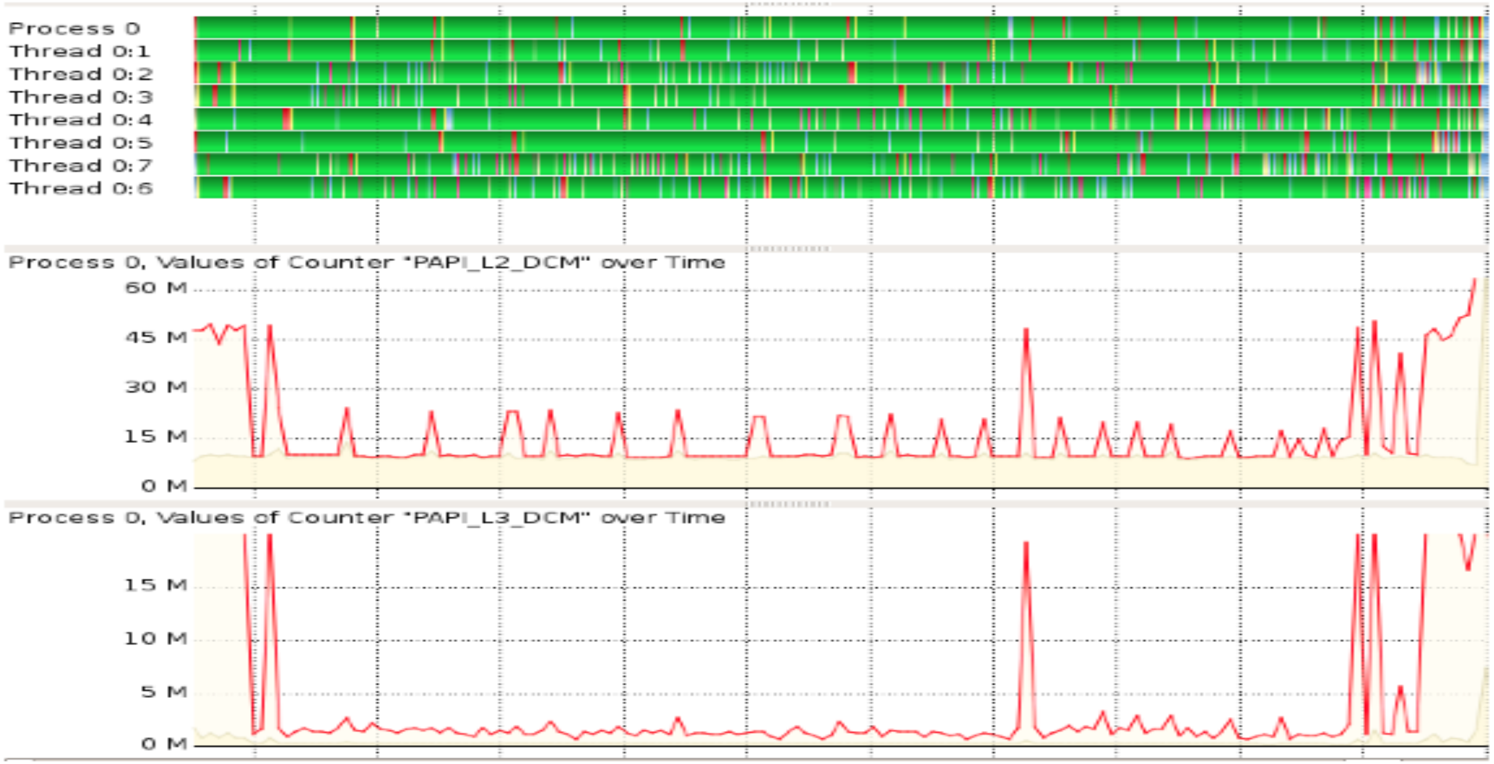
Profiling: CALU with dynamic scheduling

L2, L3 Cache misses on IBM Power 7.

m=n=5000, b=150, P = 4 x 2

L2 cache misses (max)	25M
L3 cache misses (max)	15M
Fetch task time	0.47%

CALU with dynamic scheduling and data locality



L2, L3 Cache misses on IBM Power 7.

m=n=5000, b=150, P = 4 x 2

L2 cache misses	12.5M
L3 cache misses	3.5M
Fetch task time	2.3%

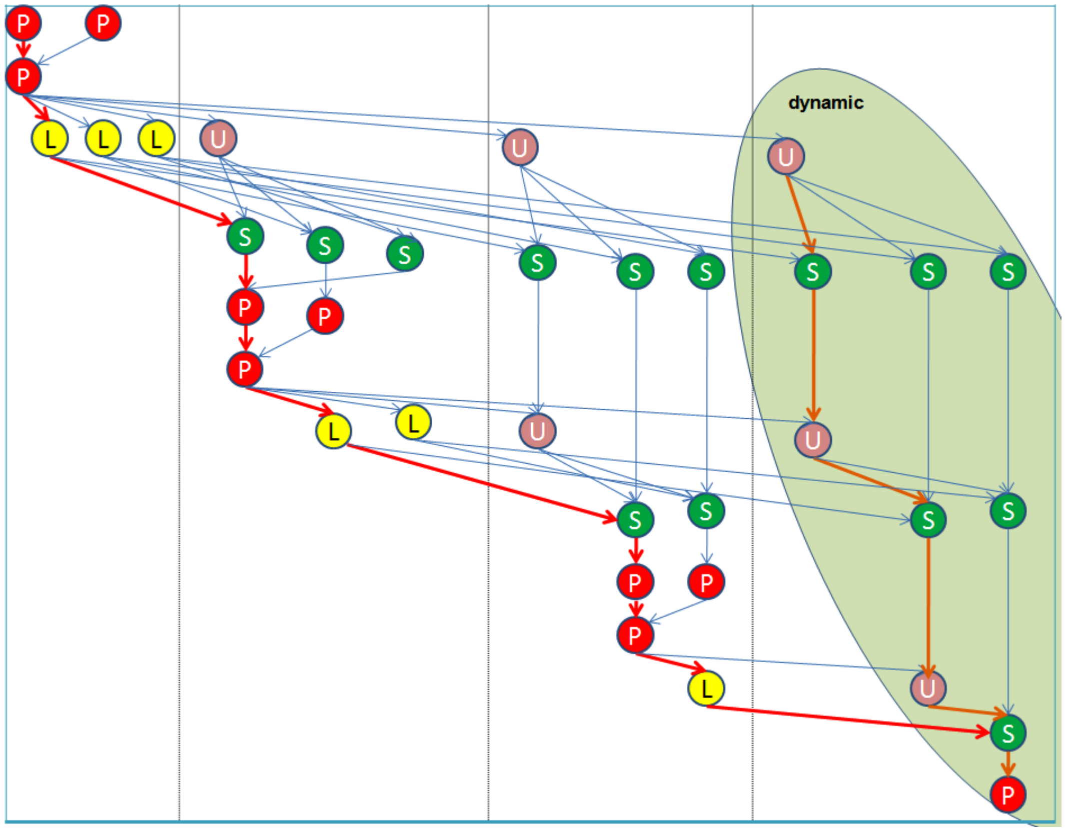
Hybrid scheduling

- Emerging complexities of multi- and mani-core processors suggest a need for self-adaptive strategies
  - One example is work stealing
- Goal:
  - Design a tunable strategy that is able to provide a good trade-off between load balance, data locality, and low dequeue overhead.
  - Provide performance consistency
- Approach: combine static and dynamic scheduling
  - Shown to be efficient for regular mesh computation [B. Gropp and V. Kale]

	Design space		
Data layout/scheduling	Static	Dynamic	Static/(%dynamic)
Block Cyclic Layout (BCL)	√	√	√
2-level Block Layout (2l-BL)	√	√	√
Column Major Layout (CM)		√	

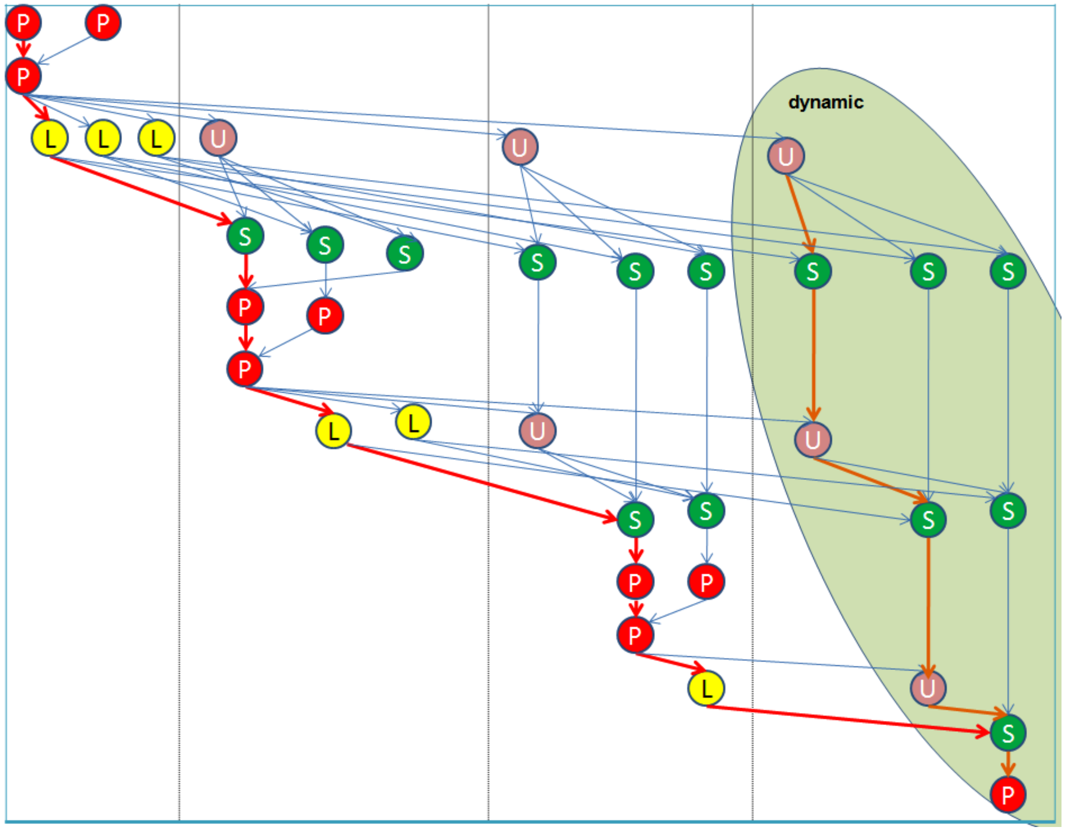
Hybrid static/dynamic scheduling

- Part of the DAG is scheduled statically
  - Using a 2D block cyclic distribution of data (tasks) to threads
- Threads execute in priority their statically assigned tasks
- When no ready task to execute, a thread picks a ready task from the dynamic part



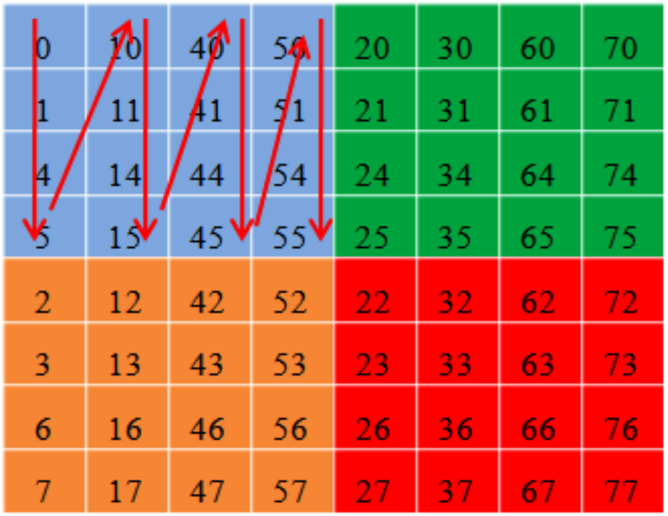
Hybrid static/dynamic scheduling (contd)

- There are two critical paths:
  - In the static part, the predefined order of execution ensures progress on the critical path
  - In the dynamic part, high priority is given to threads on the critical path

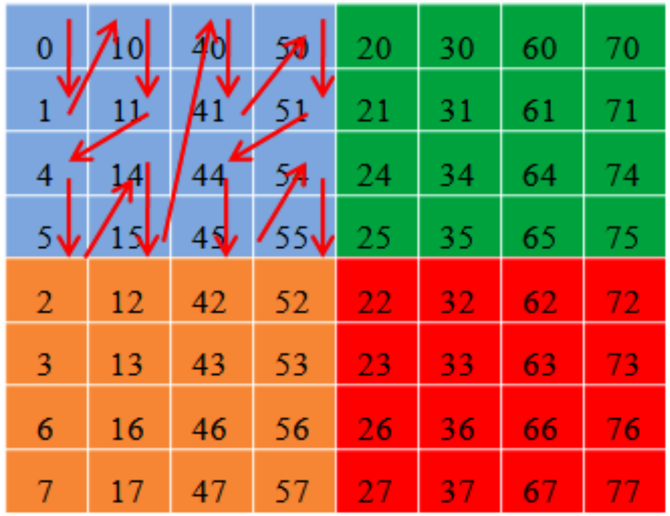


Data layout and other optimizations

- Three data distributions investigated
  - CM : Column major order for the entire matrix
  - BCL : Each thread stores contiguously (CM) the data on which it operates
  - 2l-BL : Each thread stores in blocks the data on which it operates



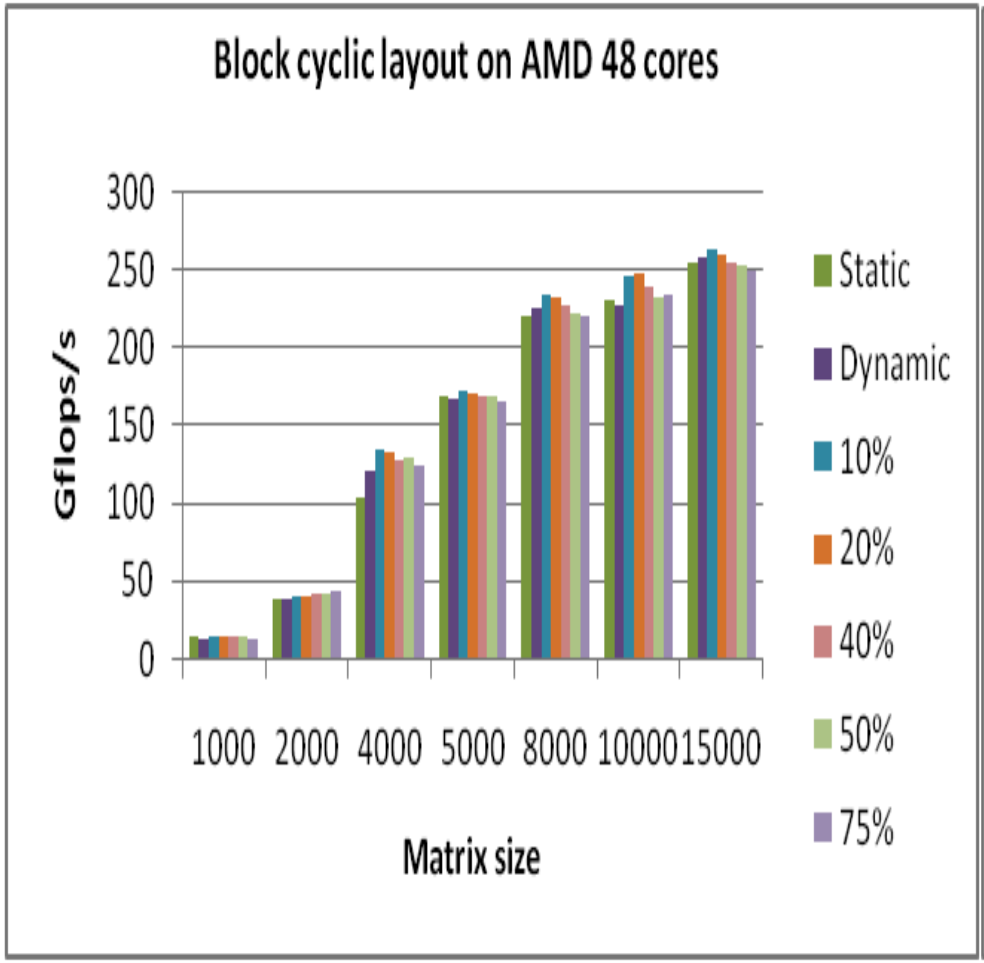
Block cyclic layout (BCL)

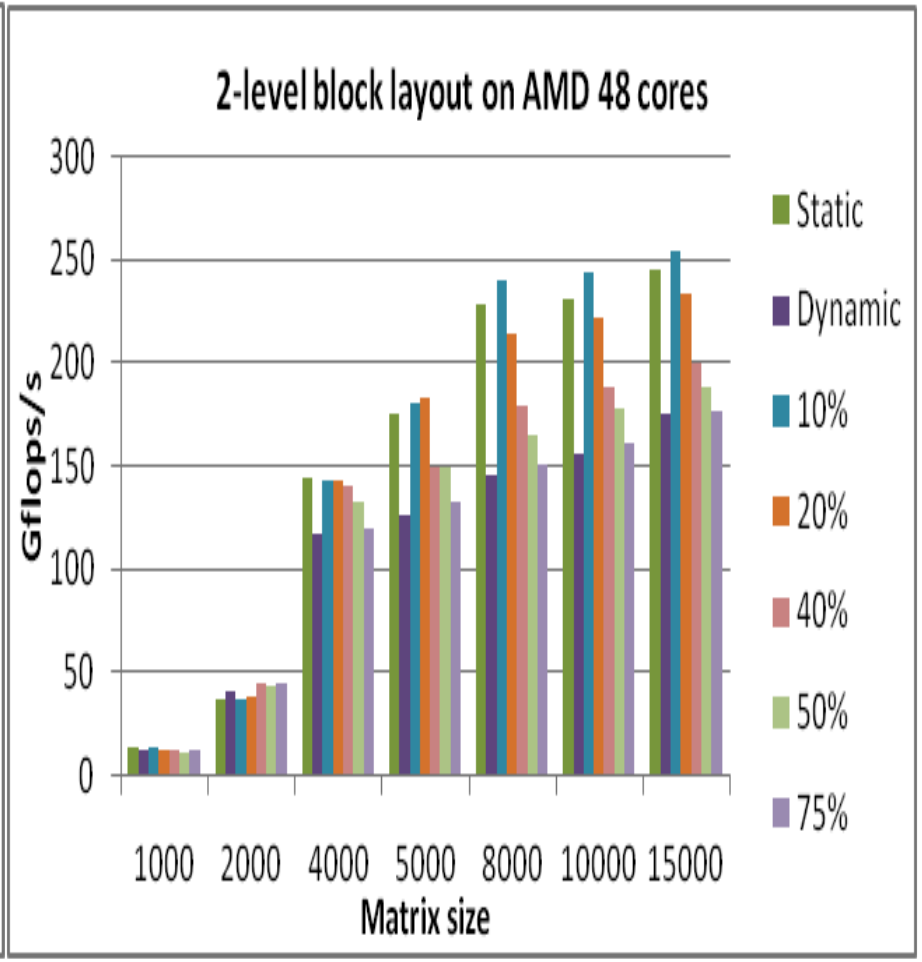


Two level block layout (2l-BL)

- And other optimizations
  - Updates (dgemm) performed on several blocks of columns (for BCL and CM layouts)

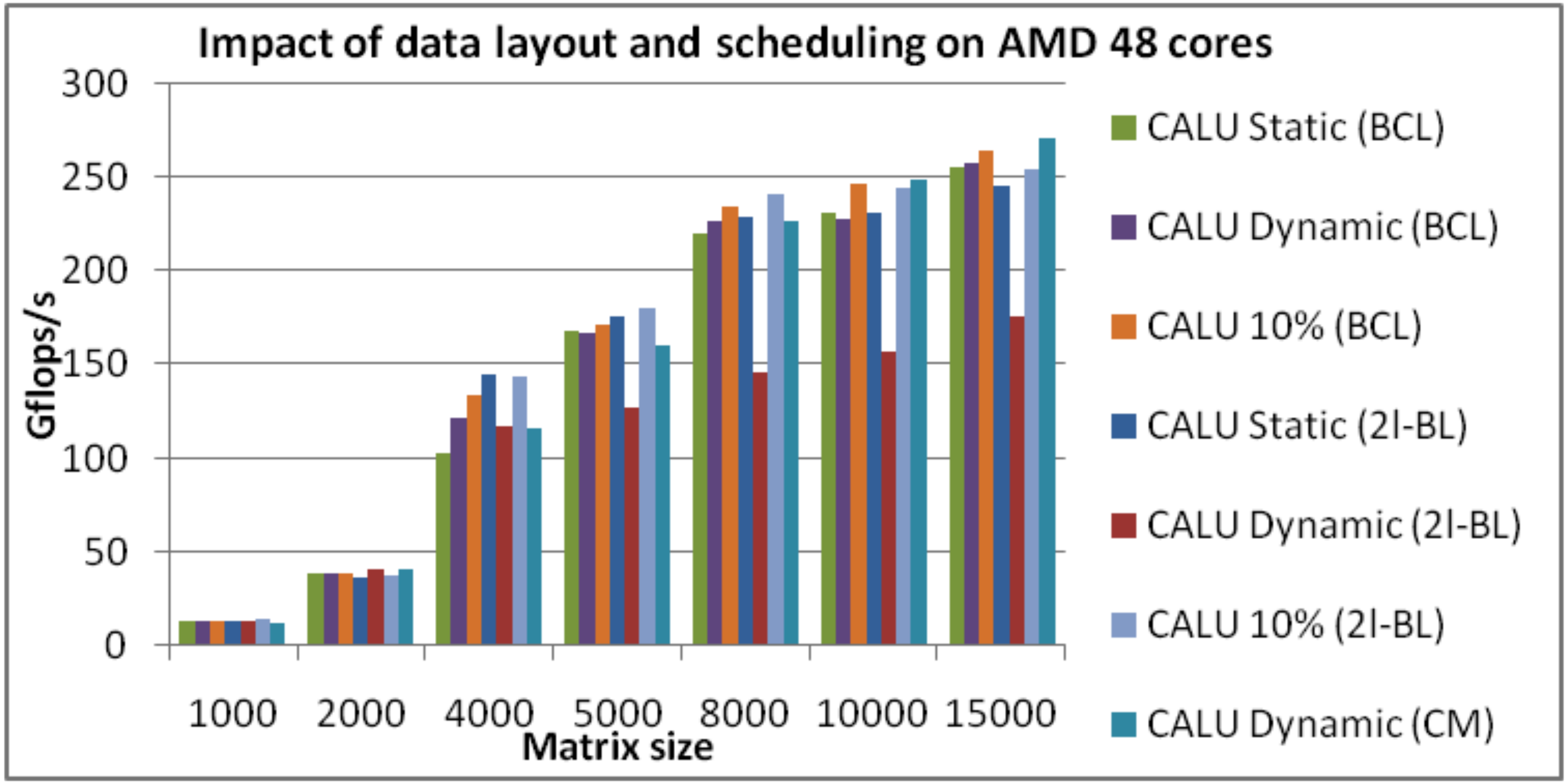
Performance of static/dynamic on multicore architectures





Eight socket, six core machine based on AMD Opteron processor (U. of Tennessee).

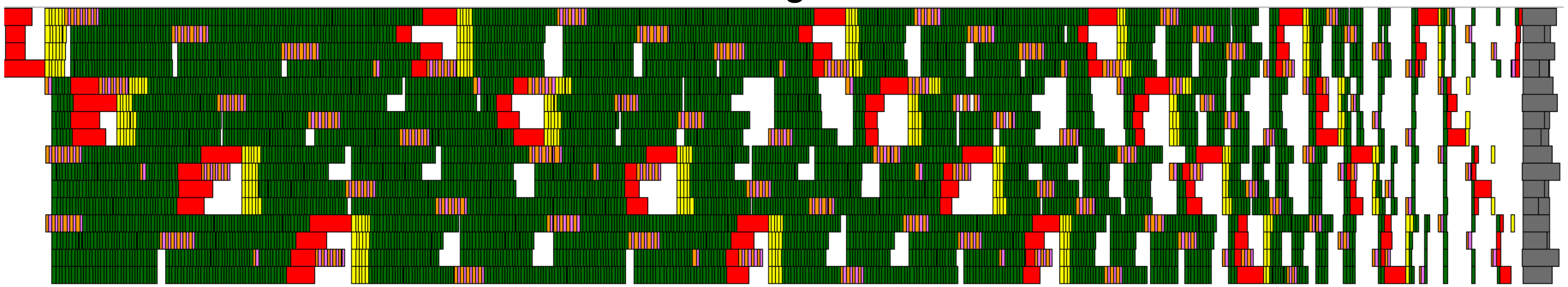
Impact of data layout



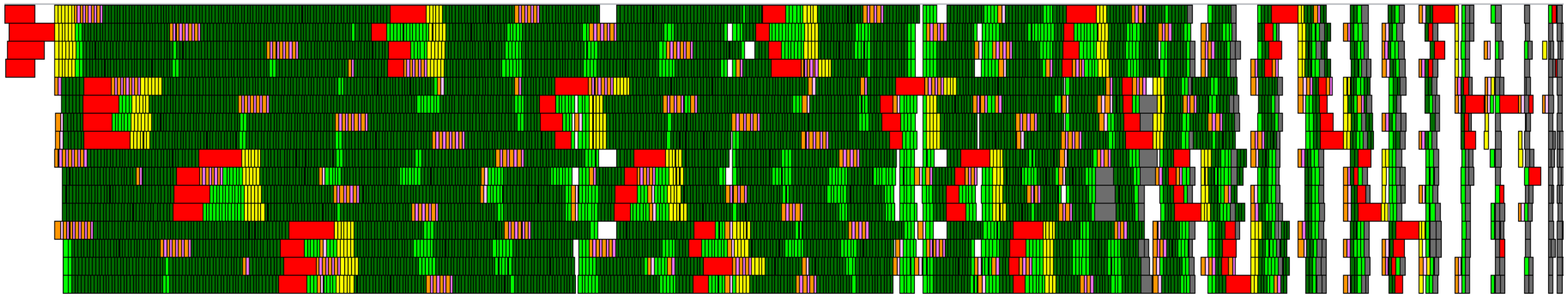
BCL : Each thread stores contiguously (CM) its data
2l-BL : Each thread stores in blocks its data

Performance of static/dynamic on multicores

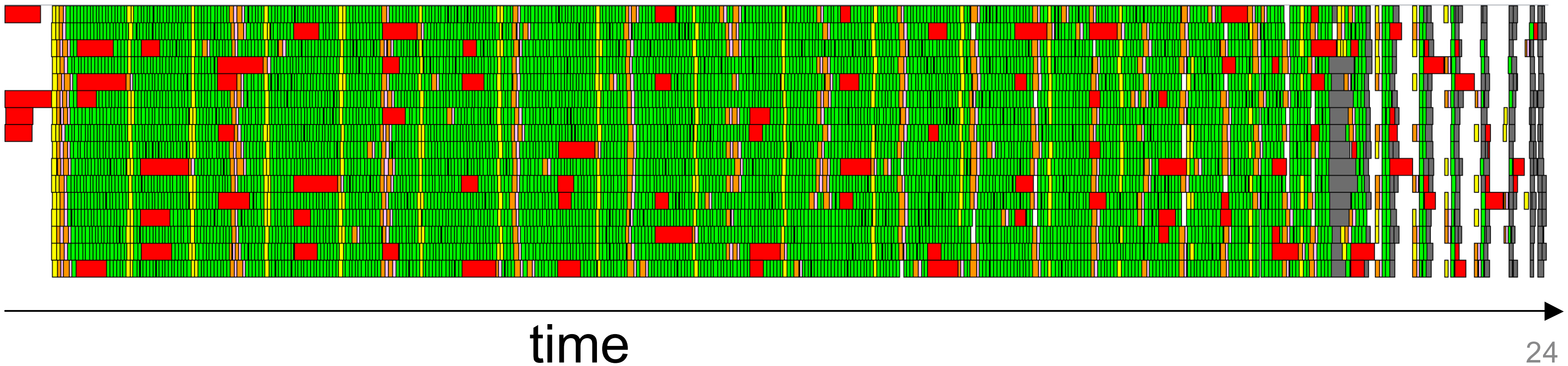
Static scheduling



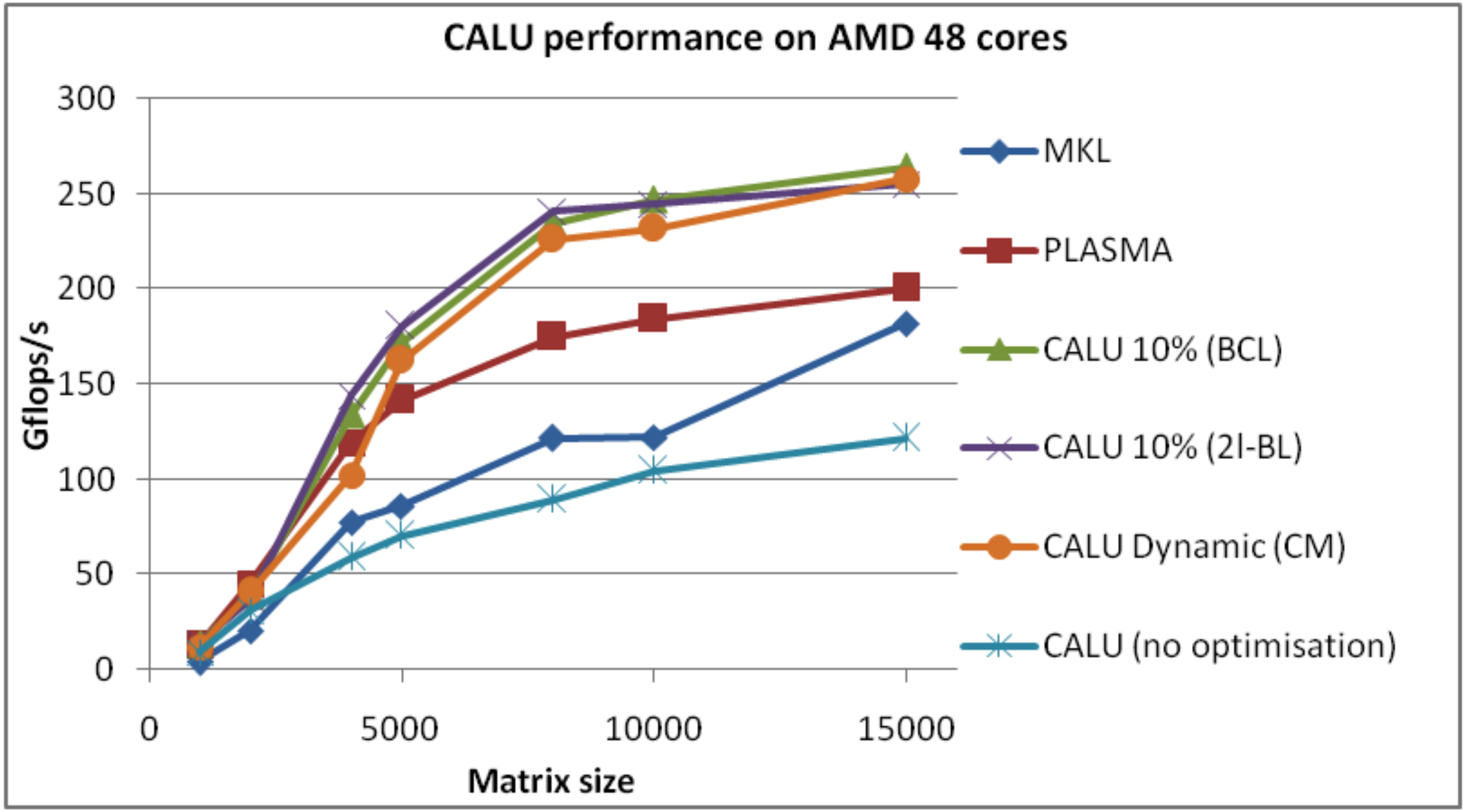
Static + **10% dynamic** scheduling



100% dynamic scheduling



Best performance of CALU on multicore architectures



- CALU 10% dynamic achieves up to 50% of the peak performance.
- Reported performance for PLASMA uses LU with incremental pivoting.

Preliminary performance model (V. Kale)

- Goal: find the breakpoint at which static scheduling induces imbalance.
- Consider the parameters:
  - f_s is the fraction of static scheduling
  - δ_i is the excess work on core i, with its max and avg values, δ_{max}, δ_{avg}
  - T_P is the time for computation to be done on P cores
- Result:

Assuming no overhead to the parallel time (eg communication), the static scheduling induces no load imbalance as long as:

$$f_s \leq 1 - \frac{\delta_{\max} - \delta_{avg}}{T_P}$$

Preliminary performance model (contd)

$$f_s \leq 1 - \frac{\delta_{\max} - \delta_{avg}}{T_P} \qquad\qquad f_d \geq \frac{\delta_{\max} - \delta_{avg}}{T_P}$$

The relation implies:

- Given δ_{max}-δ_{avg} constant
  - For a given number of processor P and increasing matrix size, the static fraction can be increased, thus avoiding scheduling overhead
  - For both weak and strong scalability, the dynamic fraction needs to be increased

- Predictions of the amplification of noise at large scale suggests that the fraction of the dynamic part will be increasing.
- Model to be continued within this collaboration.

Conclusions

- Highly efficient dense linear algebra routine
  - Based on a tunable scheduling strategy.
  - Performance of CALU on 48 cores Opteron is as good as the one reported in literature for the QR factorization (using complex reduction trees).
- Future work
  - Demonstrate the feasibility of the hybrid scheduling for other operations.
  - Develop a performance model to guide the choice of the fraction of the static/dynamic parts of the scheduler.