# Hybrid static/dynamic scheduling for already optimized dense matrix factorization

Simplice Donfack, Laura Grigori,
INRIA, France

Bill Gropp, Vivek Kale
UIUC, USA

Joint Laboratory for Petascale Computing, INRIA-UIUC

## Plan

- Brief introduction of communication avoiding methods
- Lightweight 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

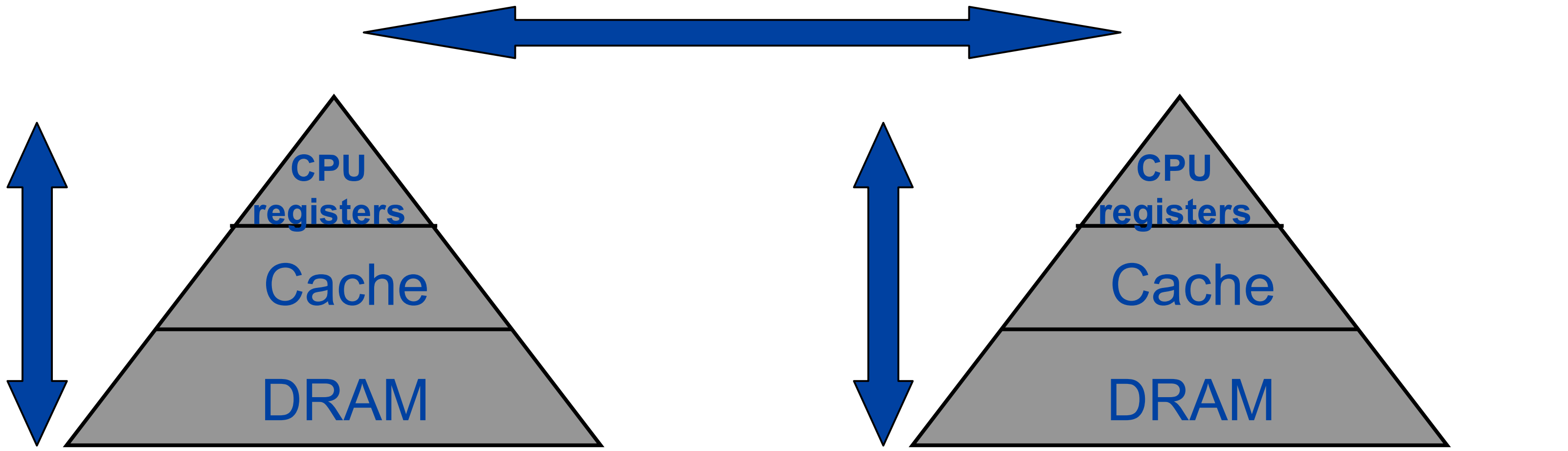
- Time to move data >> time per flop

Running time =
#flops * **time_per_flop** +
#words_moved / **bandwidth** +
#messages * **latency**

Improvements per year

| DRAM | Network |
|---|---|
| **23%** | **26%** |
| **5%** | **15%** |

- Gap steadily and exponentially growing over time

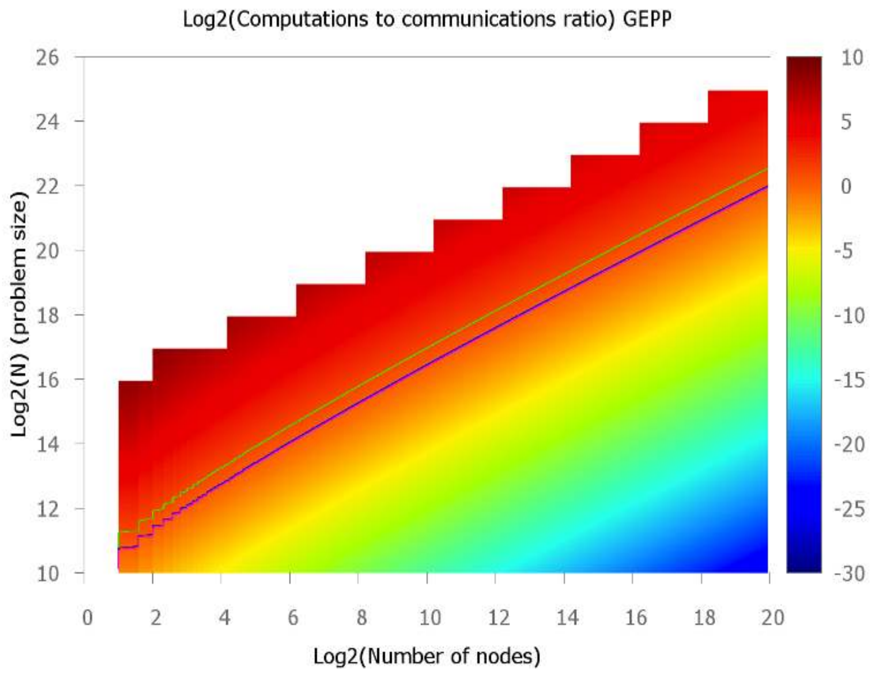


## Previous work on reducing communication

- Tuning
  - Overlap communication and computation, at most a factor of 2 speedup

- Ghosting
  - Store redundantly data from neighboring processors for future computations

- Scheduling
  - Cache oblivious algorithms for linear algebra
    - Gustavson 97, Toledo 97, Frens and Wise 03, Ahmed and Pingali 00
  - Block algorithms for linear algebra
    - ScaLAPACK, Blackford et al 97



Courtesy of M. Jacquelin

## Algorithms and lower bounds on communication

- Goals for algorithms in dense linear algebra
  - Minimize #words_moved = $\Omega$ (#flops/ $M^{1/2}$ ) = $\Omega$ ( $n^2$ / $P^{1/2}$ )
  - Minimize #messages = $\Omega$ (#flops/ $M^{3/2}$ ) = $\Omega$ ( $P^{1/2}$ )
  - Allow redundant computations (preferably as a low order term)

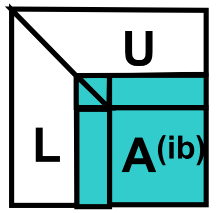
- LAPACK and ScaLAPACK
  - Mostly suboptimal
- Recursive cache oblivious algorithms
  - Minimize bandwidth, not latency, sometimes more flops (3x for QR)

- CA algorithms for dense linear algebra
  - Minimize both bandwidth and latency
  - Optimal CAQR, CALU introduced in 2008 by Demmel, Hoemmen, LG, Langou, Xiang
  - General bounds proven in 2011 by Ballard, Demmel, Holtz, Schwartz

# LU factorization (as in ScaLAPACK pdgetrf)

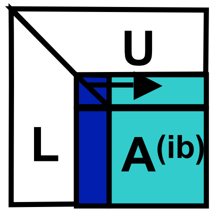
LU factorization on a $P = P_r \times P_c$ grid of processors

```
For i = 1 to n-1 step b
    A^(ib) = A(ib:n, ib:n)
```



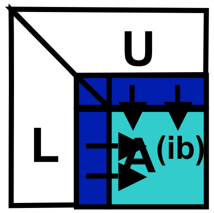
(1) Compute panel factorization (pdgetf2) $O(n \log_2 P_r)$

- find pivot in each column, swap rows



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

- swap rows at left and right

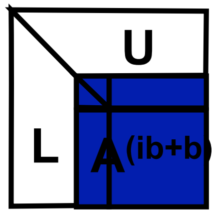


(3) Compute block row of U (pdtrsm) $O(n/b \log_2 P_c)$

- 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



## Factorizations that require pivoting

- Known pivoting techniques that minimize communication lead to unstable factorizations
- Requires new tournament pivoting scheme (LU, RRQR)

- Consider a block algorithm that factors an n-by-n matrix A.

$$A = \begin{array}{cc} \overbrace{\phantom{A_{11}}}^{b} & \overbrace{\phantom{A_{12}}}^{n-b} \end{array} \\ A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{matrix} \}\, b \\ \}\, n-b \end{matrix}, \text{ where } \quad W = \begin{pmatrix} A_{11} \\ A_{21} \end{pmatrix}$$

- 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.

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

# Tournament pivoting for a tall skinny matrix

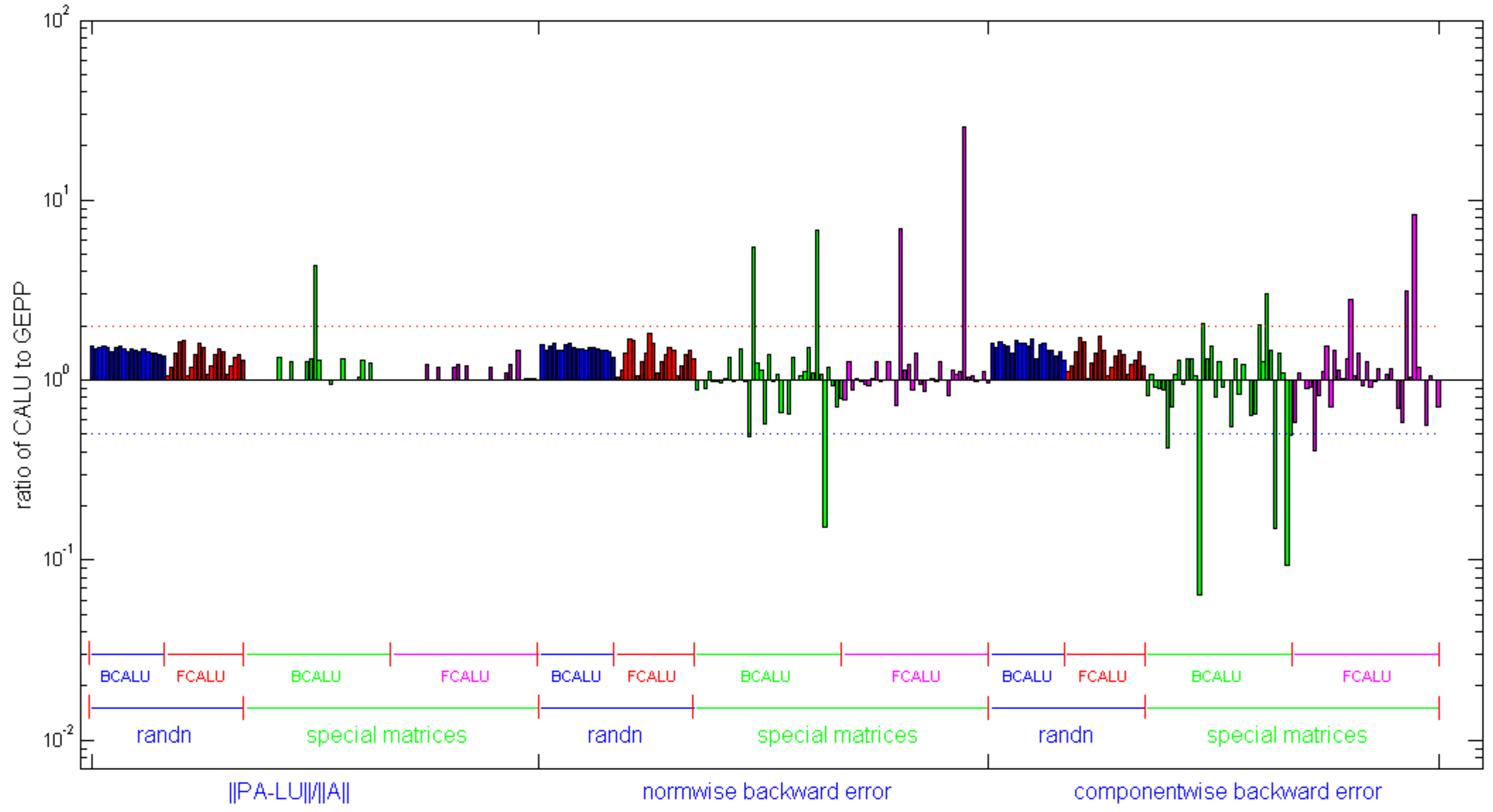
$$\begin{tikzcd}
P_0 & W_0 = \begin{pmatrix} 2 & 4 \\ 0 & 1 \\ 2 & 0 \\ 1 & 2 \end{pmatrix} = \Pi_0 L_0 U_0 & \Pi_0^T W_0 = \begin{pmatrix} 2 & 4 \\ 2 & 0 \end{pmatrix} \arrow[r] & \overline{W}_0 = \begin{pmatrix} 2 & 4 \\ 2 & 0 \\ 4 & 1 \\ 2 & 0 \end{pmatrix} = \overline{\Pi}_0 \overline{L}_0 \overline{U}_0 & \overline{\Pi}_0^T \overline{W}_0 = \begin{pmatrix} 4 & 1 \\ 2 & 4 \end{pmatrix} \arrow[r] & \underline{W}_0 = \begin{pmatrix} 4 & 1 \\ 2 & 4 \\ 4 & 2 \\ 1 & 4 \end{pmatrix} = \underline{\Pi}_0 \underline{L}_0 \underline{U}_0 & \underline{\Pi}_0^T \underline{W}_0 = \begin{pmatrix} 4 & 1 \\ 1 & 4 \end{pmatrix} \\
P_1 & W_1 = \begin{pmatrix} 2 & 0 \\ 0 & 0 \\ 4 & 1 \\ 1 & 0 \end{pmatrix} = \Pi_1 L_1 U_1 & \Pi_1^T W_1 = \begin{pmatrix} 4 & 1 \\ 2 & 0 \end{pmatrix} \arrow[ur] & & & & \\
P_2 & W_2 = \begin{pmatrix} 0 & 1 \\ 1 & 4 \\ 0 & 0 \\ 0 & 2 \end{pmatrix} = \Pi_2 L_2 U_2 & \Pi_2^T W_2 = \begin{pmatrix} 1 & 4 \\ 0 & 2 \end{pmatrix} \arrow[r] & \overline{W}_2 = \begin{pmatrix} 1 & 4 \\ 0 & 2 \\ 4 & 2 \\ 0 & 2 \end{pmatrix} = \overline{\Pi}_2 \overline{L}_2 \overline{U}_2 & \overline{\Pi}_2^T \overline{W}_2 = \begin{pmatrix} 4 & 2 \\ 1 & 4 \end{pmatrix} \arrow[uur] & & \\
P_3 & W_3 = \begin{pmatrix} 2 & 1 \\ 0 & 2 \\ 1 & 0 \\ 4 & 2 \end{pmatrix} = \Pi_3 L_3 U_3 & \Pi_3^T W_3 = \begin{pmatrix} 4 & 2 \\ 0 & 2 \end{pmatrix} \arrow[ur] & & & &
\end{tikzcd}$$

Good pivots for factorizing W

$$\xrightarrow{\text{time}}$$

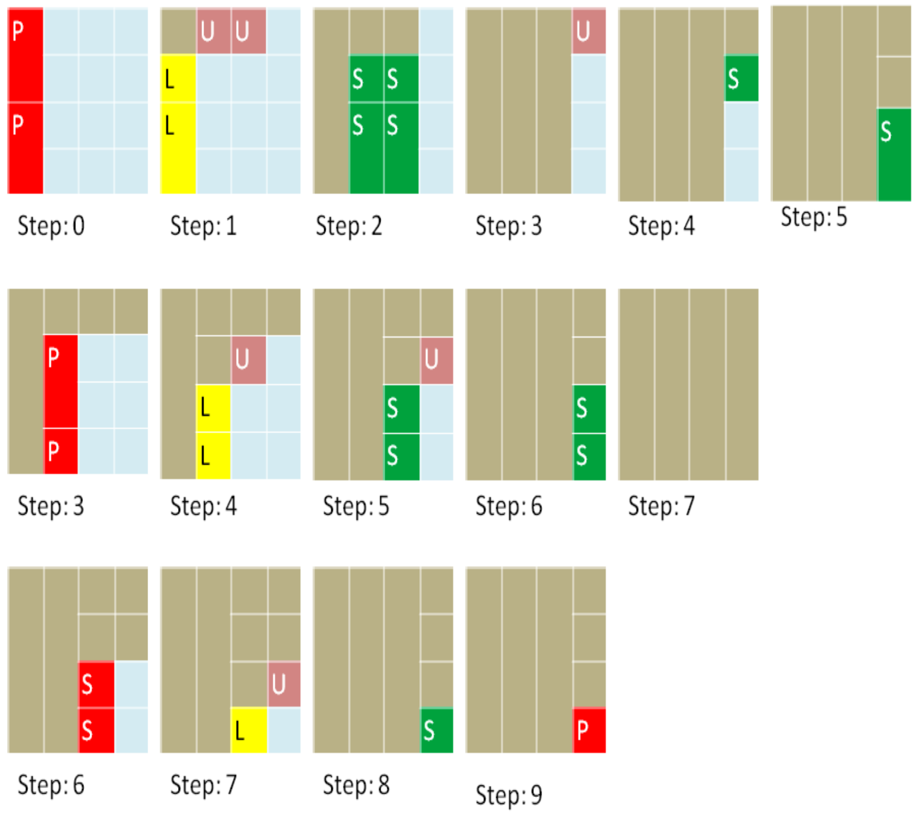
# Stability of CALU (experimental results)

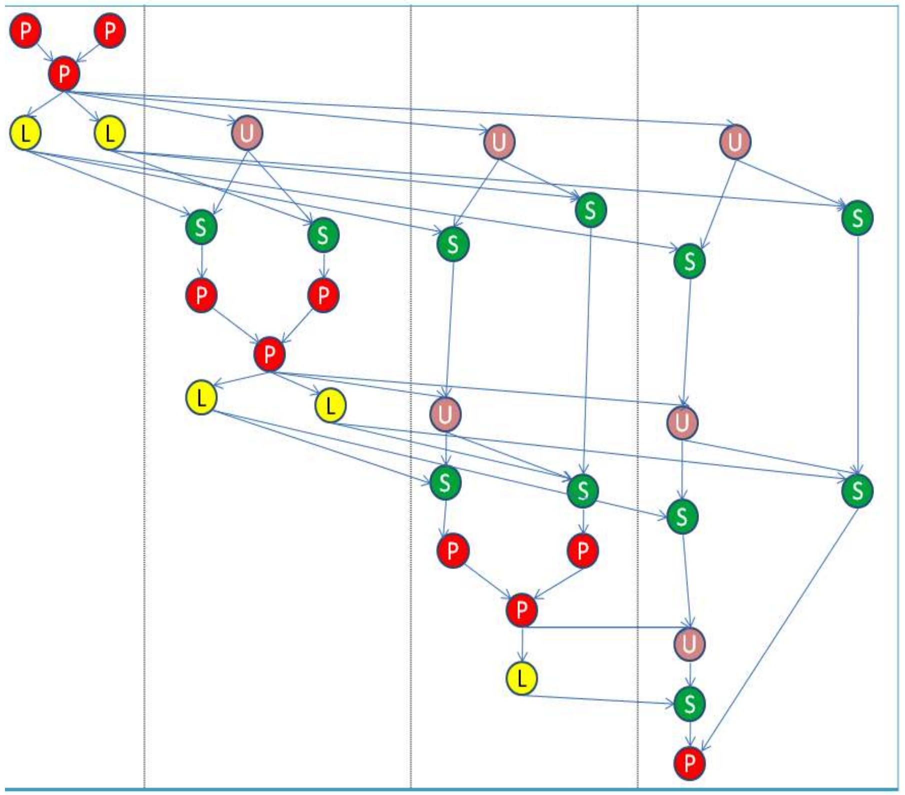
- Results show ||PA-LU||/||A||, normwise and componentwise backward errors, for random matrices and special ones
  - See [LG, Demmel, Xiang, 2011, SIMAX] for details
  - BCALU denotes binary tree based CALU and FCALU denotes flat tree based CALU



# CALU and its task dependency graph

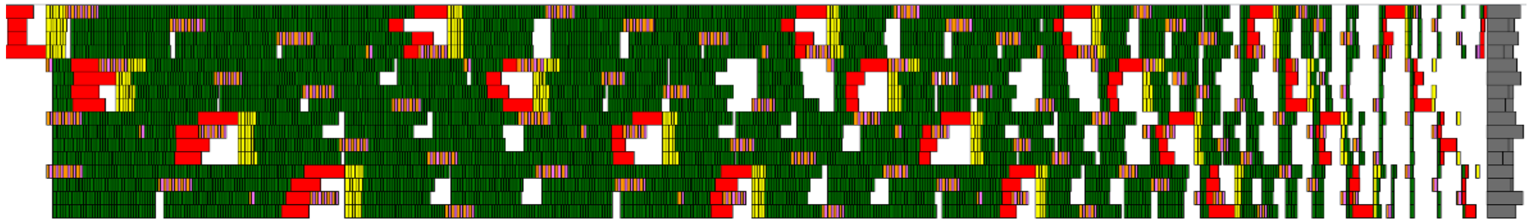
- The matrix is partitioned into blocks of size T x b.
- The computation of each block is associated with a task.





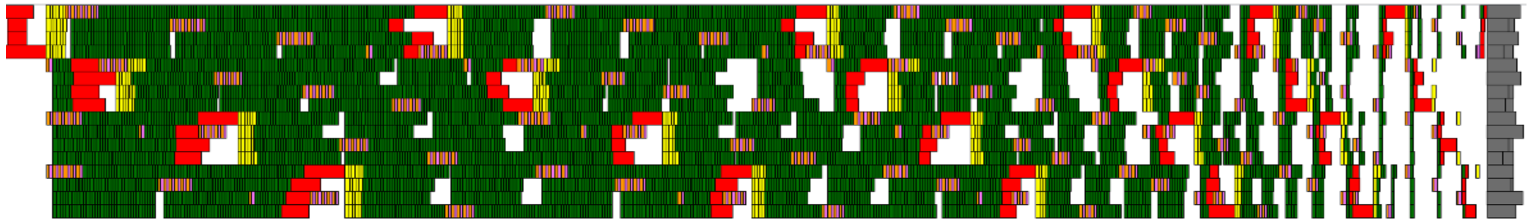
## Scheduling CALU's Task Dependency Graph

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

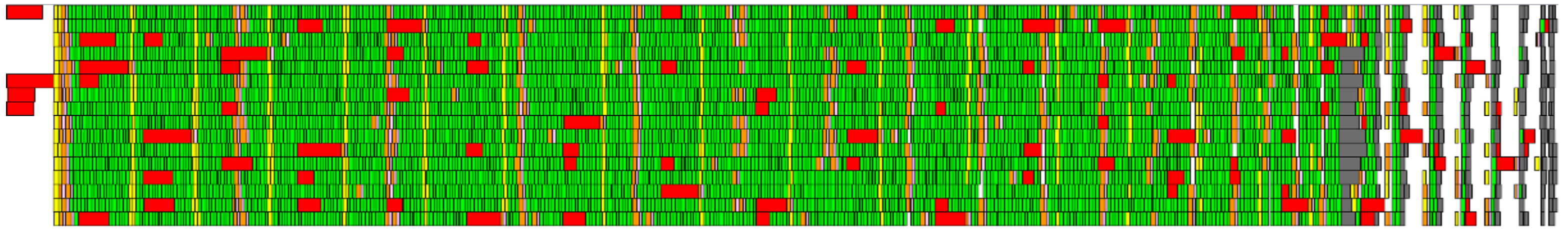


# Scheduling CALU's Task Dependency Graph

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

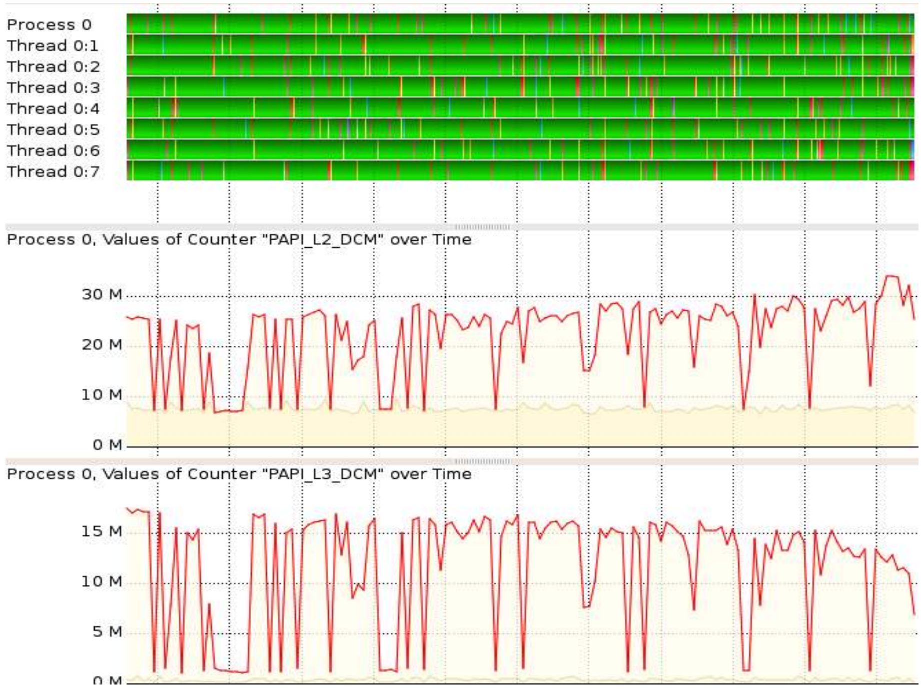


- Dynamic scheduling
  - + Keeps cores busy
  - - Poor usage of data locality
  - - Can lead to large overhead



# Profiling: CALU with dynamic scheduling

Dynamic scheduling

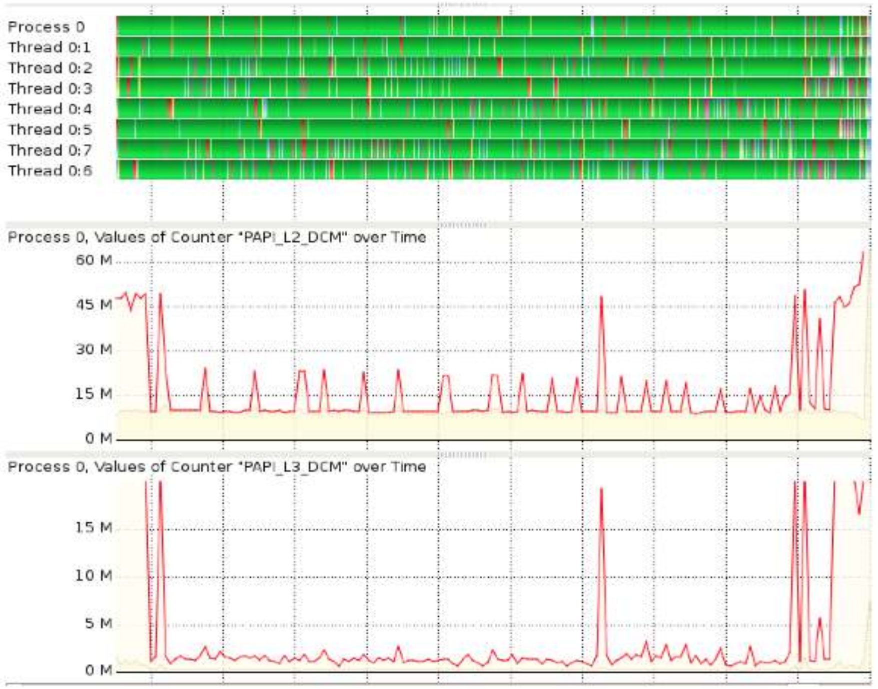


L2, L3 Cache misses on IBM Power 7.

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

| | |
|---|---|
| L2 cache misses | 25M |
| L3 cache misses | 15M |
| Fetch task time | 0.47% |

Dynamic scheduling with 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% |

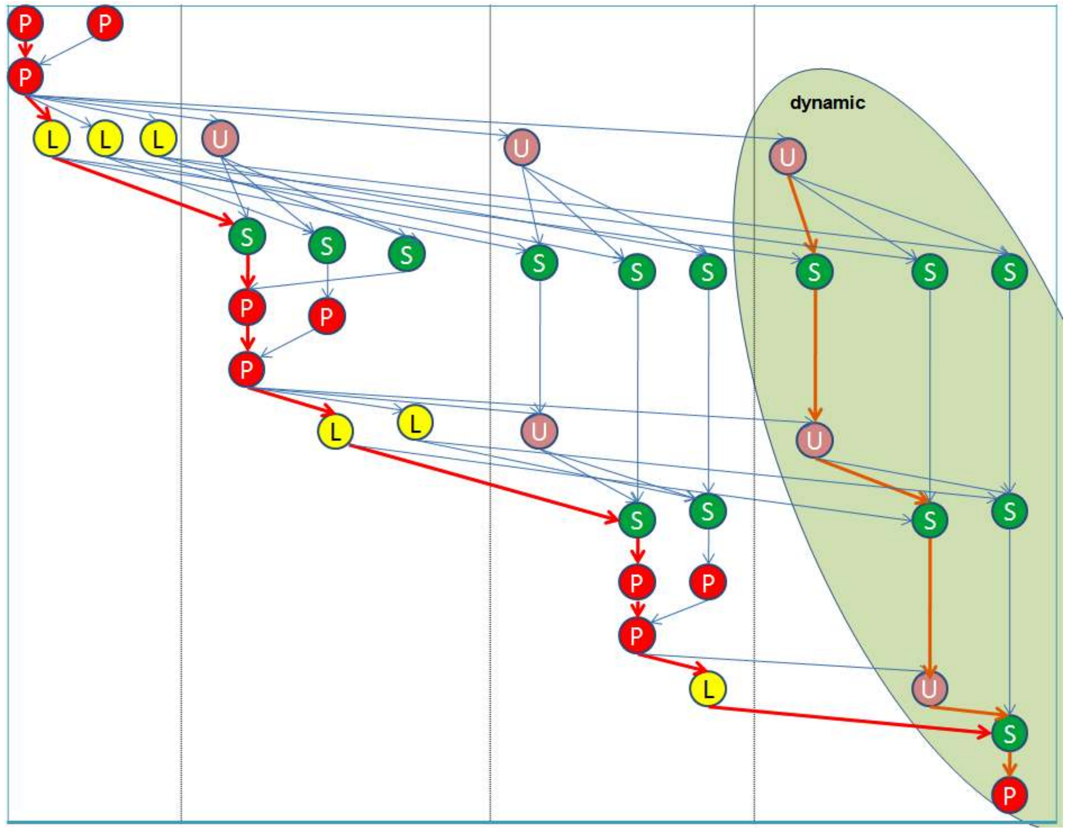
## Lightweight scheduling

- 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) | | √ | |

## Lightweight scheduling: hybrid static/dynamic approach

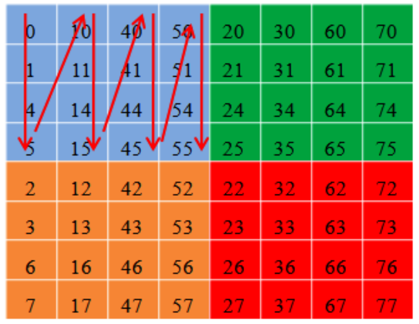
- Part of the DAG is scheduled statically
  - Using a 2D block cyclic distribution of data (tasks) to threads
- A thread executes in priority its statically assigned tasks
- When no task is ready, a thread picks a ready task from the dynamic part



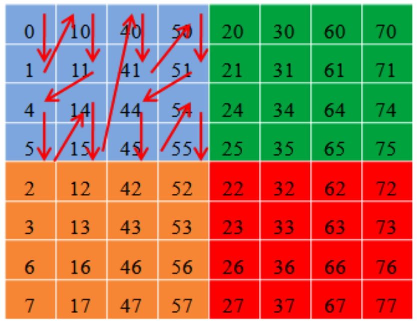
## Impact of data layout

Data layouts:

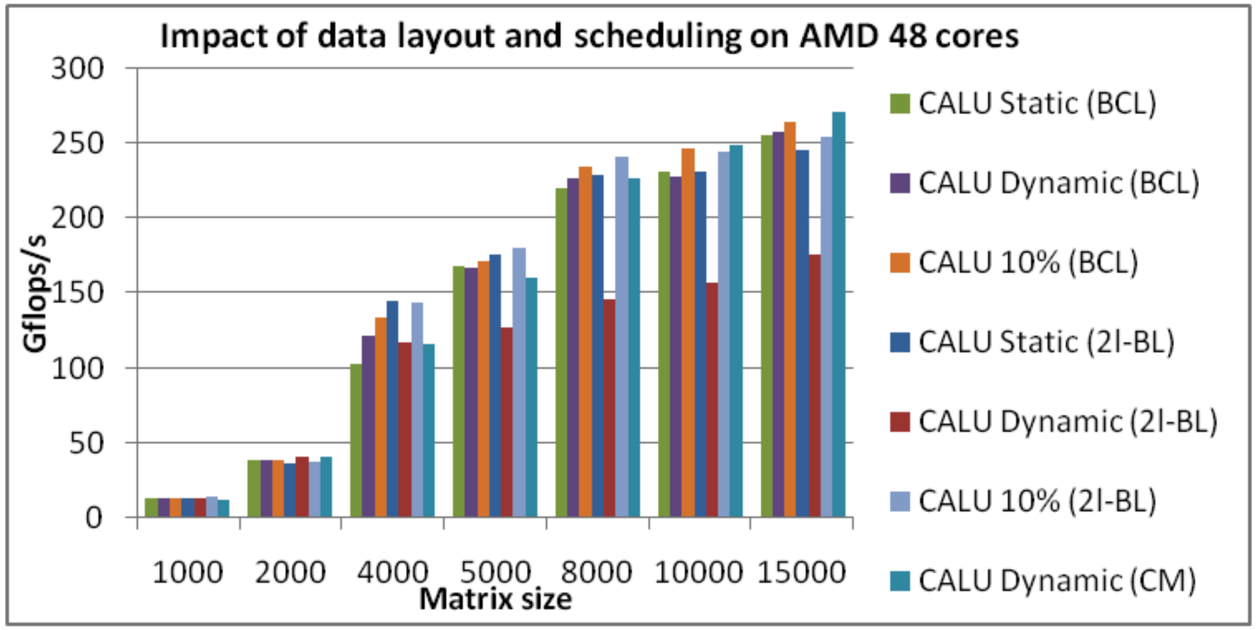
- CM : Column major order
- BCL : Each thread stores its data using CM
- 2l-BL : Each thread stores its data in blocks



Block cyclic layout (BCL)

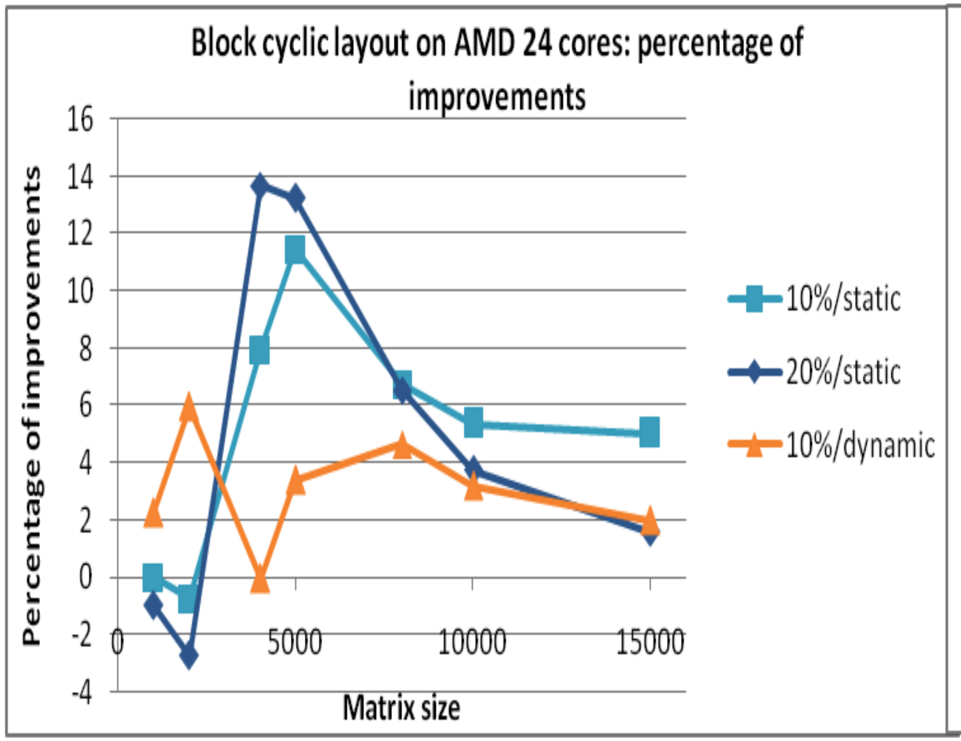


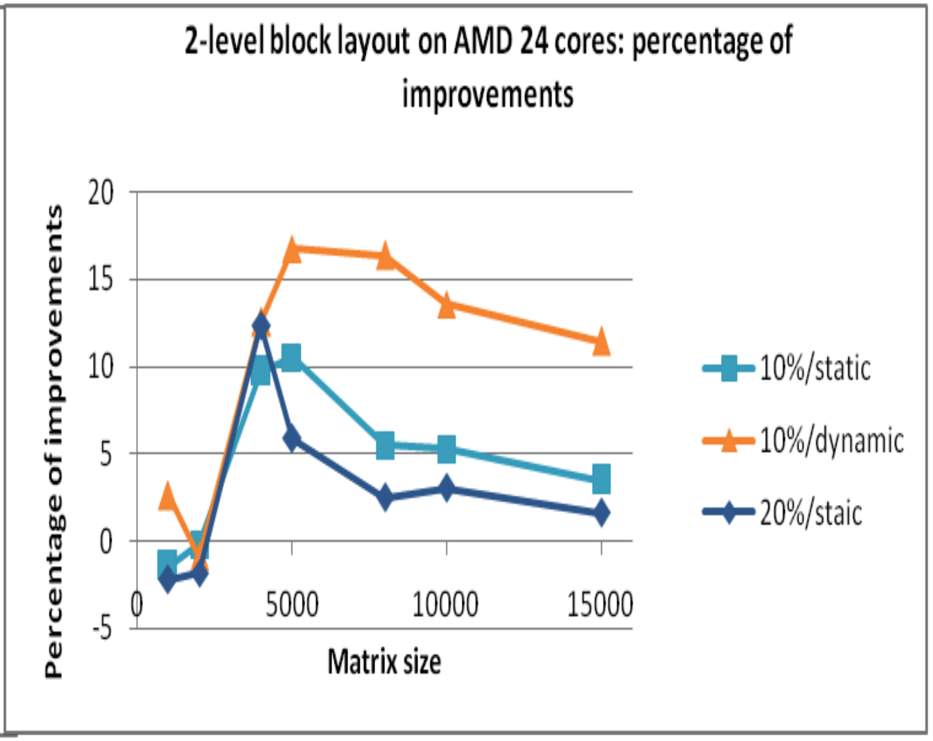
Two level block layout (2l-BL)



Four socket, twelve cores machine based on AMD Opteron processor (U. of Tennessee).

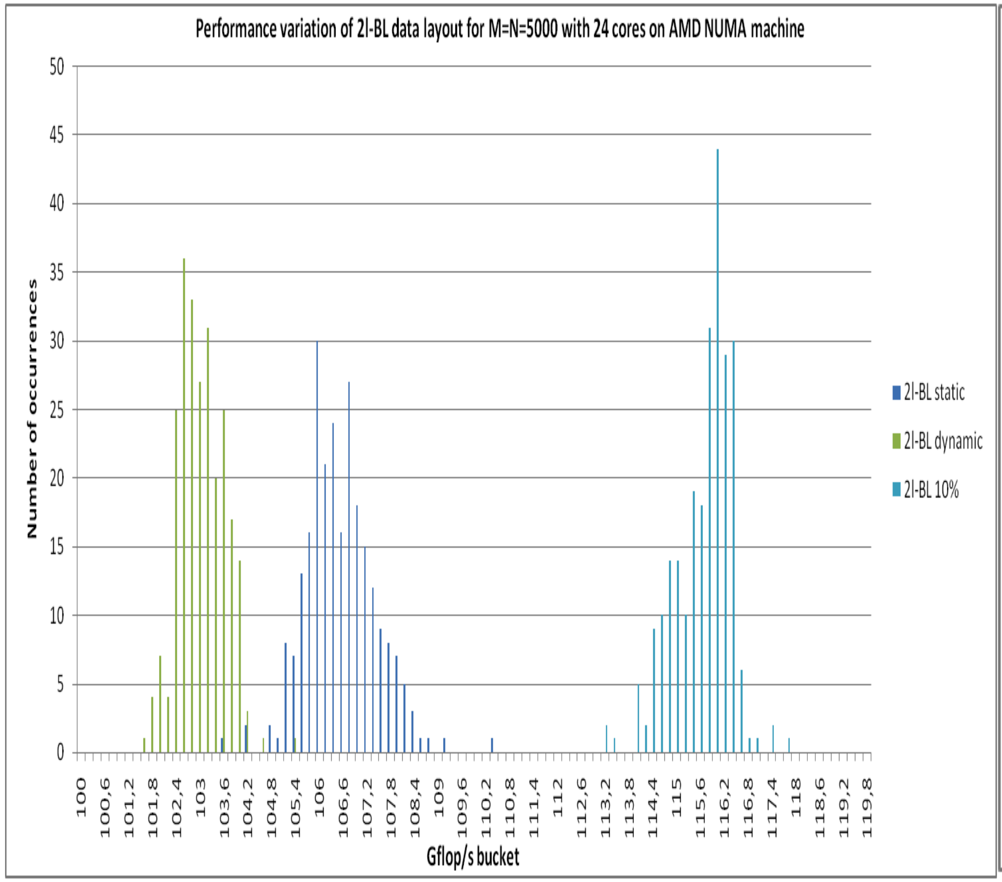
## Improvement with respect to static and dynamic scheduling

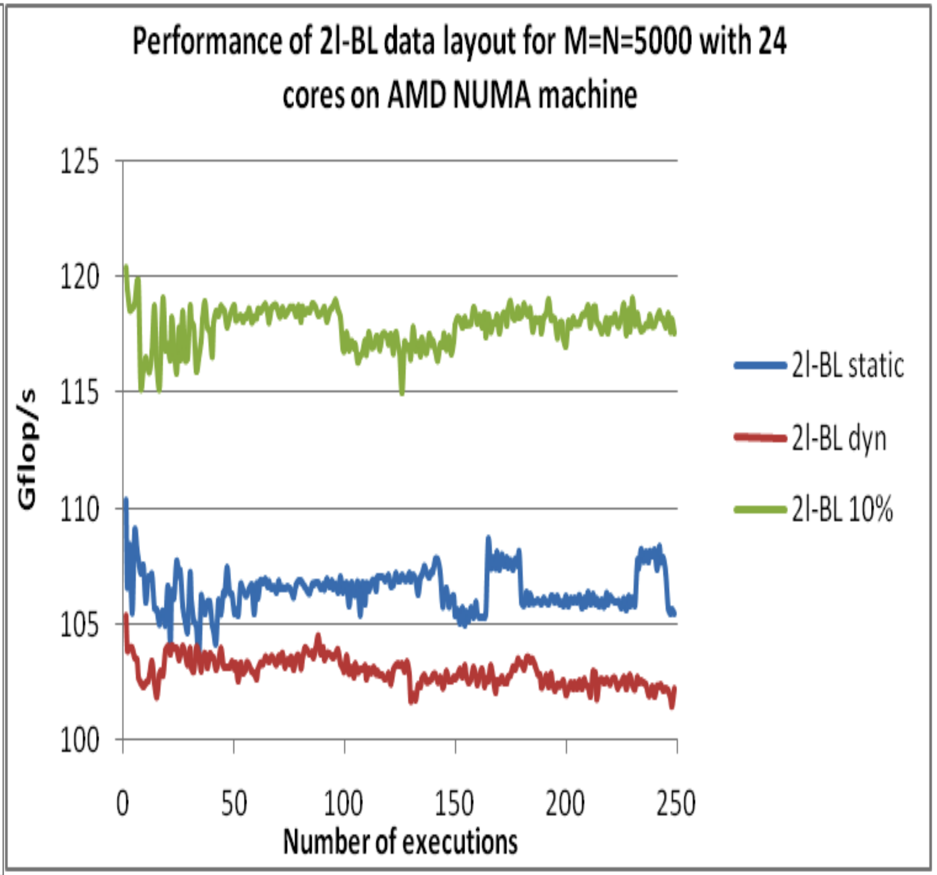




Four socket, twelve cores machine based on AMD Opteron processor (U. of Tennessee).

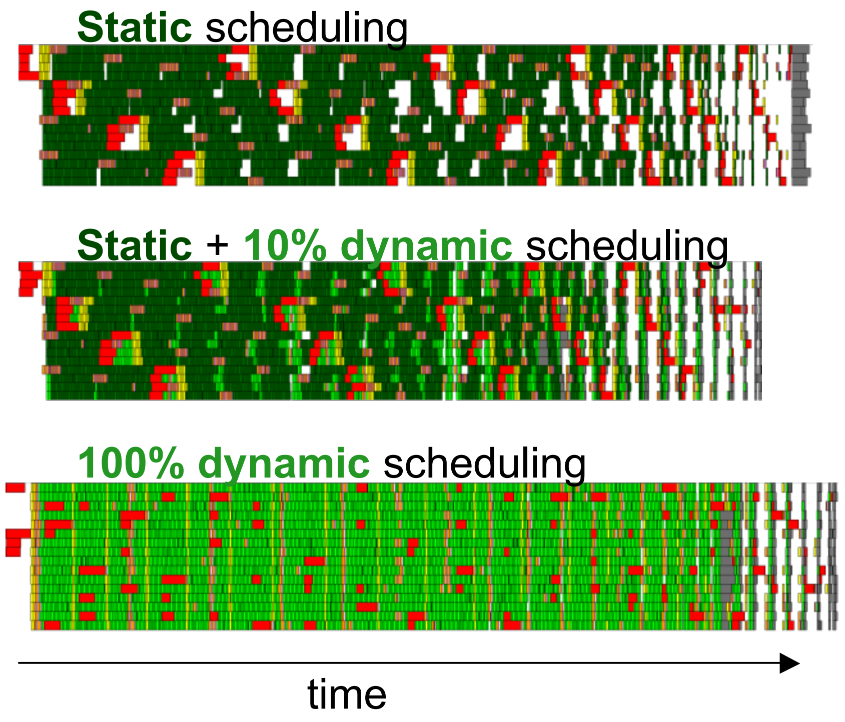
# Performance variations for 250 runs

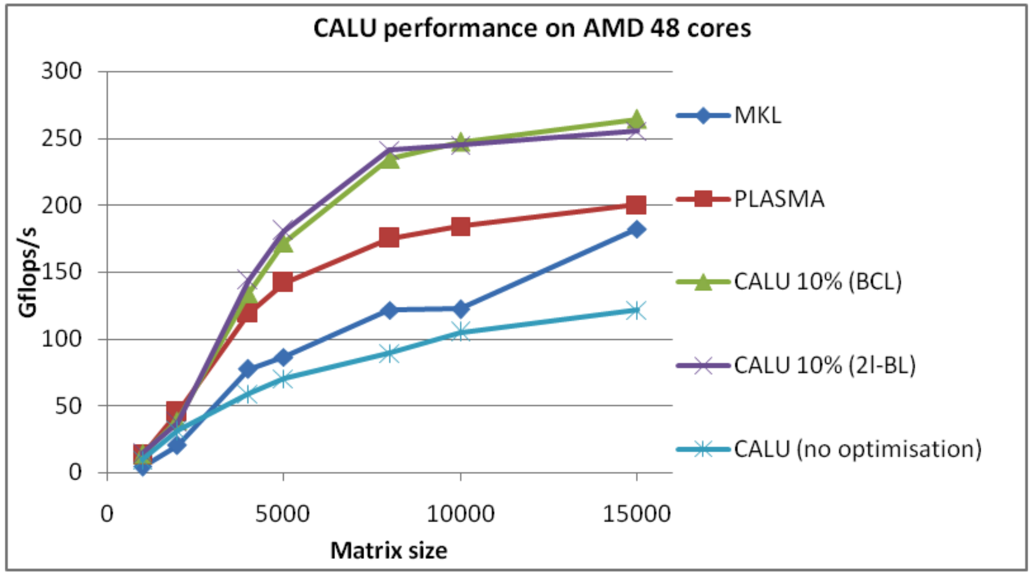




- Using 10% dynamic for our single node tests, we not only get high-performance, but also performance consistency
- Our solution addresses the noise amplification problem, where localized noise can amplify and create large bottlenecks at 10000+ nodes

# Best performance of CALU on multicore architectures





- CALU 10% dynamic achieves up to 60% of the peak performance
- Reported performance for PLASMA uses LU with incremental pivoting

## Performance model: first results

- Find the breakpoint at which static scheduling induces load imbalance
- Consider the parameters
  - $f_s$ is the fraction of static scheduling
  - $\delta_i$ is the excess work on core i
  - $\delta_{total}$ is the sum of excess work across all cores
  - $T_P$ is the time for computation to be done on P cores
- Assuming no overhead to the parallel time (eg communication), the static scheduling induces no load imbalance as long as

$$f_s \leq 1 - \frac{\delta_{total}}{T_P}$$

## Performance model: first results

$$f_s \le 1 - \frac{\delta_{total}}{T_P} \qquad\qquad f_d \ge \frac{\delta_{total}}{T_P}$$

- Given $\delta_{total}$ constant
  - For a given number of processor P and increasing matrix size, the static fraction can be increased, thus avoiding scheduling overhead
  - For 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

## 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 performance reported in literature for the QR factorization (using complex reduction trees)
- Future work
  - Demonstrate the feasibility of the lightweight scheduling for other operations
  - Develop a detailed theoretical analysis to guide the choice of the percentage dynamic in the scheduler
  - Apply the theoretical analysis of lightweight scheduling  time complexity to CALU, CAQR