# EVALUATING THE IMPACT OF INTEL KNL MEMORY SETTINGS ON PERFORMANCE THROUGH CASE STUDIES

I. Masliah

28-03-2017

# KNL on PLAFRIM

## Current configurations :

- Kona01 : flat / quadrant
- Kona02 : cache / quadrant
- Kona03 : hybrid / quadrant
- Kona04 : cache / SNC-4

## What we have seen :

For memory bound problems, the flat memory mode is always more efficient

## This is also true for compute bound problems (if it fits in MCDRAM)

# Direct allocations in MCDRAM

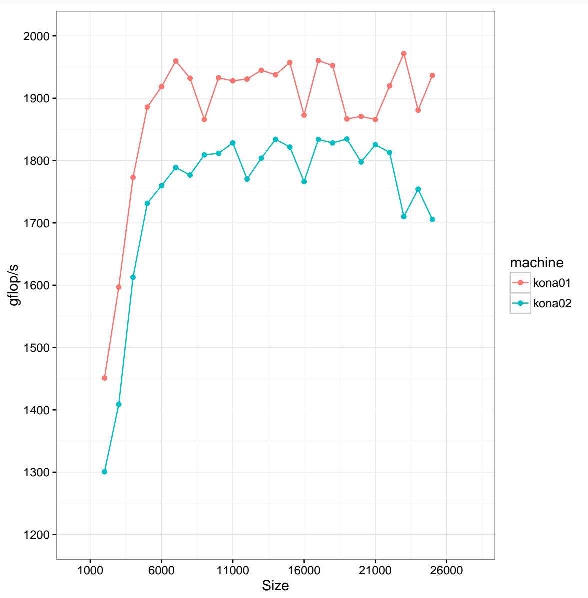
## Need to download and install memking

- Available at https://github.com/memkind
- Provides a special malloc, a memory allocator in C++ and Fortran attributes

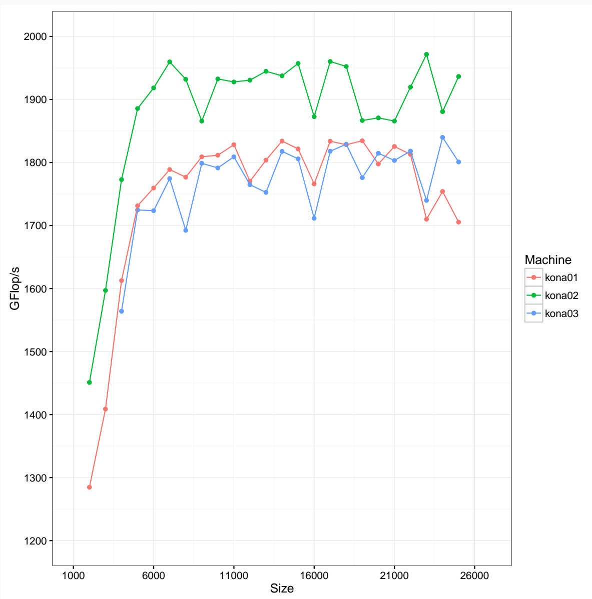
```
#include <vector>
#include <hbw_allocator.h>

using T = double;

std::vector<T, hbw::allocator<T>> A(m*n);
std::vector<T, hbw::allocator<T>> B(m*n);
std::vector<T, hbw::allocator<T>> C(m*n);

cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
            m , n, k, alpha , A.data(), lda, B.data(), ldb, beta, C.data(),  ldc);
```

# MATRIX PRODUCT

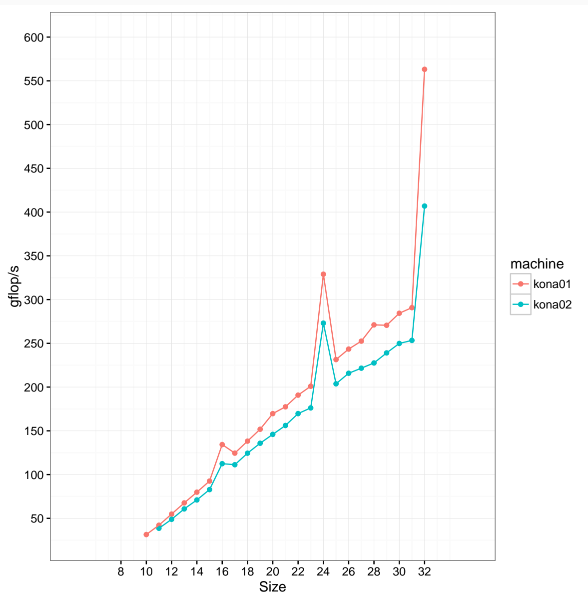
# Scaling matrix product (square matrices)



# Scaling matrix product (square matrices)



# batched matrix product (100 000)



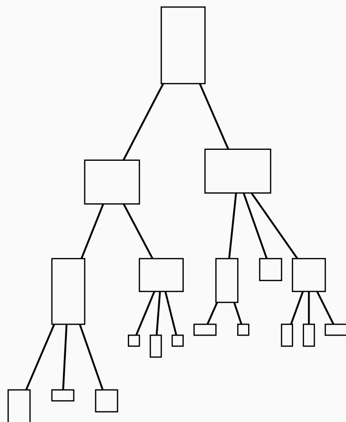
# QR_MUMPS

AUTHORS OF THIS STUDY: EMMANUEL AGULLO, ALFREDO BUTTARI, MIKKO BYCKLING, ABDOU GUERMOUCHE, IAN MASLIAH

# The Multifrontal QR method

The original multifrontal method by Duff & Reid '83 can be extended to QR factorization of sparse matrices.
This method is guided by a graph called *elimination tree*:

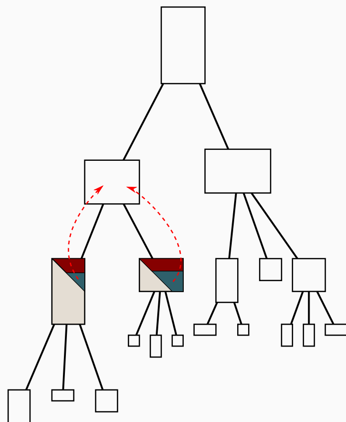
- each node is associated with a relatively small dense matrix called frontal matrix (or front) containing k pivots to be eliminated along with all the other coefficients concerned by their elimination.



# The Multifrontal QR method

The tree is traversed in topological order (i.e., bottom-up) and, at each node, two operations are performed:

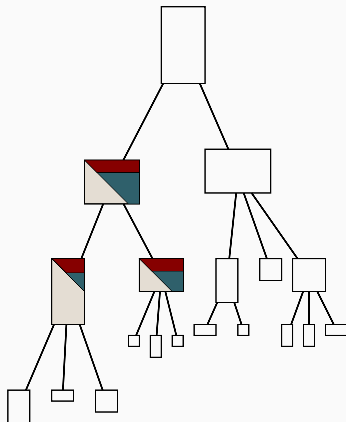
- assembly: coefficients from the original matrix associated with the pivots and *contribution blocks* produced by the treatment of the child nodes are stacked to form the frontal matrix.



# The Multifrontal QR method

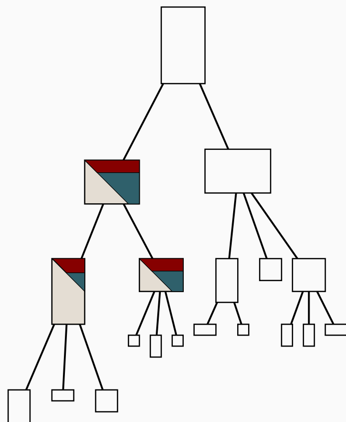
The tree is traversed in topological order (i.e., bottom-up) and, at each node, two operations are performed:

- assembly: coefficients from the original matrix associated with the pivots and *contribution blocks* produced by the treatment of the child nodes are stacked to form the frontal matrix.
- factorization: the $k$ pivots are eliminated through a complete dense QR factorization of the frontal matrix. As a result we get:
  - part of the global $R$ and $Q$ factors.
  - a triangular *contribution block* that will be assembled into the father's front.



# The Multifrontal QR method

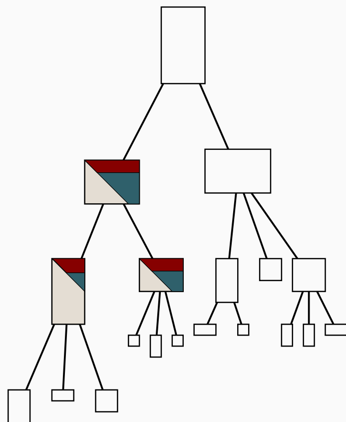
Typically two sources of parallelism are exploited in the multifrontal method



# The Multifrontal QR method

Typically two sources of parallelism are exploited in the multifrontal method

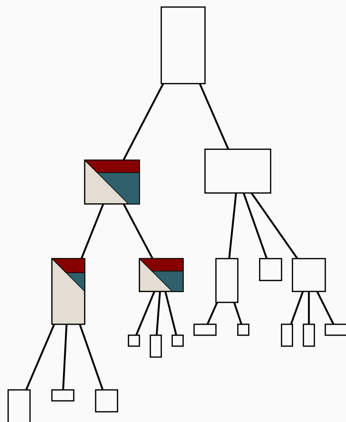
- tree-level parallelism: frontal matrices located in independent branches in the tree can be processed in parallel.



# The Multifrontal QR method

Typically two sources of parallelism are exploited in the multifrontal method

- tree-level parallelism: frontal matrices located in independent branches in the tree can be processed in parallel.
- node-level parallelism: large frontal matrices factorization may be performed in parallel by multiple threads.



# qr_mumps Experimental Conditions

Matrices from the UF SParse Matrix Collection:

| Mat. name | m | n | nz | op. count (Gflop) | peak mem (GB) |
|---|---|---|---|---|---|
| spal_004 | 10203 | 321696 | 46168124 | 27059 | 23.3 |
| TF17 | 38132 | 48630 | 586218 | 38209 | 12.8 |
| n4c6-b6 | 104115 | 51813 | 728805 | 97304 | 35.6 |
| lp_nug30 | 52260 | 379350 | 1567800 | 171051 | 83.4 |
| TF18 | 95368 | 123867 | 1597545 | 194472 | 78.1 |

- Factorization step only
- Implementation over the StarPU runtime system

# Tuning the KNL system for `qr_mumps`

- Important memory requirements
- Large number of dynamic memory allocations

## The Hardware

- Cache mode: Flat, Cache, Hybrid
- Clustering mode: All-to-All, Quadrant, Hemisphere, SNC2, SNC4

## The Operating System

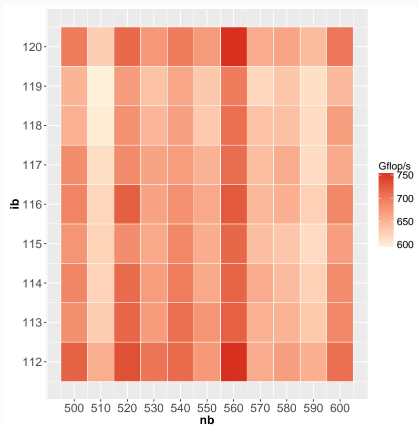
- Huge pages : Transparent Huge Page, TBB
  - Standard page size : 4KB
  - Huge page size : 2MB, 1GB
  - Number : freely settable
- Memory Allocator : default, TBB

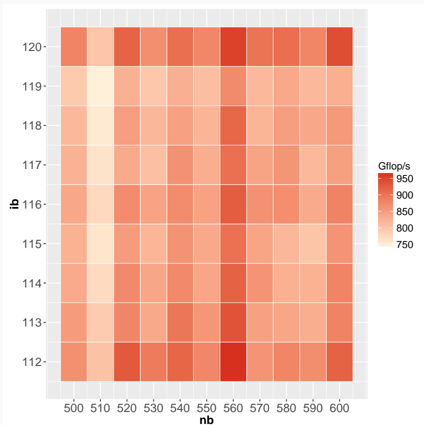
# Tuning the KNL system for qr_mumps

- Important memory requirements
- Large number of dynamic memory allocations

| Test machines : | |
|---|---|
| System 1 (KNL64) | Intel(R) Xeon Phi(TM) CPU 7210 - 64 cores @1.3 GHz |
| System 2 (KNL68) | Intel(R) Xeon Phi(TM) CPU 7250 - 68 cores @1.4 GHz |
| System 3 (BDW) | Intel(R) Xeon(R) E5 2697v5 - 2 sockets, 18 cores @2.3 Ghz |
| **KNL Hardware settings :** | |
| Clustering mode | quadrant |
| MCDRAM mode | cache |
| **Operating system/memory settings :** | |
| Operating system | RHEL 7.2 |
| Memory allocator | TBB : scalable allocator, Explicit Hugepages (8000) |
| THP | always active |
| Hugepage size | 2MB |
| **Libraries settings :** | |
| Compiler | Intel Parallel Studio 2017, Update 1 |
| BLAS library | Intel Math Kernel Library, 2017 Update 1 |
| qr_mumps | 2.0 |
| StarPU/scheduler | trunk (rev.19630)/ws |

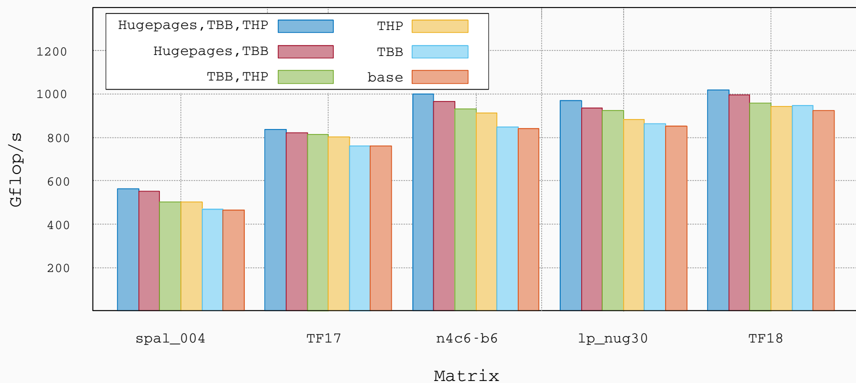
# Tuning block sizes on KNL




Impact of block size for fronts (KNL64) of size $16384 \times 8192$ (*left*) and $20480 \times 16384$ (*right*)

# Tuning memory settings for Multifrontal QR (KNL64)



# Energy efficiency and performance

| | Gflop/s | | | Gflop/s/watt | |
|---|---|---|---|---|---|
| Matrix | BDW | KNL64 | KNL68 | BDW | KNL68 |
| spal_004 | 605.35 | 562.21 | 579.43 | 1.31 | 1.91 |
| TF17 | 674.51 | 837.55 | 954.50 | 1.49 | 2.88 |
| lp_nug30 | 730.05 | 970.23 | 1057.18 | 1.65 | 3.13 |
| n4c6-b6 | 759.01 | 1001.79 | 1076.38 | 1.62 | 3.12 |
| TF18 | 761.72 | 1018.61 | 1092.40 | 1.56 | 3.03 |

# PASTIX

AUTHORS OF THIS STUDY: MATHIEU FAVERGE, GREGOIRE PICHON, PIERRE RAMET, JEAN ROMAN

# Problem to solve

## Problem: solve $Ax = b$

- Cholesky: factorize $A = LL^T$ (symmetric pattern $(A + A^T)$ for $LU$)
- Solve $Ly = b$
- Solve $L^T x = y$

## Sparse Direct Solvers: PaStiX approach

1. Order unknowns to minimize the fill-in
2. Compute a symbolic factorization to build $L$ structure
3. Factorize the matrix in place on $L$ structure
4. Solve the system with forward and backward triangular solves

# Pastix Experimental Conditions

## Set of matrices

- Subset of large matrices from SuiteSparse collection, around 1 million unknowns each

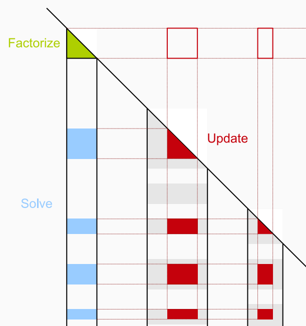
## PaStiX

- Factorization step only
- Implementation over the parsec runtime system
- Blocking sizes from 160 to 320 on low $flops/nnz_L$ ratio
- Blocking sizes from 320 to 640 on high $flops/nnz_L$ ratio

# Numerical Factorization

## Algorithm to eliminate the block column $k$

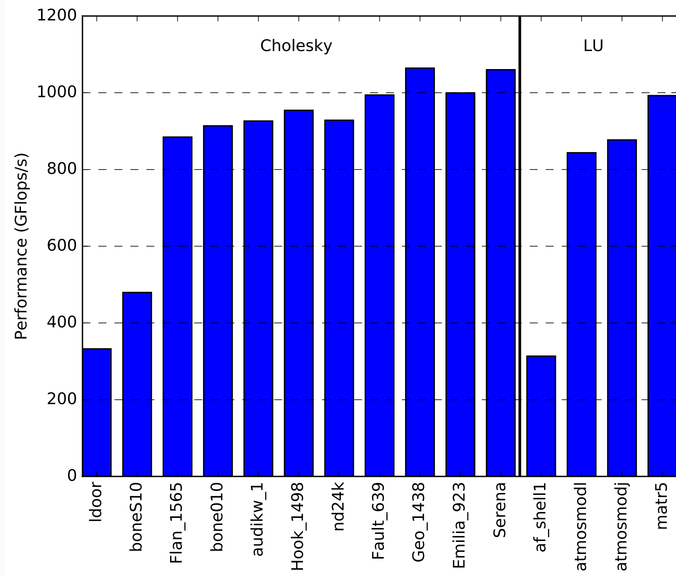
1. Factorize the diagonal block (POTRF/GETRF)
2. Solve off-diagonal blocks in the current column (TRSM)
3. Update the trailing matrix with the column's contribution (GEMM)




## How to do it

- 1D updates per block of columns for lower level of elimination tree
- 2D updates $\approx$ Dense factorization for higher levels

# Performance on the KNL architecture



# Conclusions

## KNL Memory modes

- If a problem fits in MCDRAM, it is usuallt better to use flat mode
- Manual allocations in MCDRAM are possible with hbm
- Tested problems do not fit in flat memory so we stick to quadrant
- Some interesting material for KNL : Prace

## On sparse direct methods

- Modern runtime systems work great for implementing complex applications on single-node, accelerated systems.
- For more details on `qr_mumps` for KNL see[1]
- For more details on PaStiX, ask Mathieu Faverge for the SIAM CSE 2017 talk

[1] E. Agullo et al. *Achieving high-performance with a sparse direct solver on Intel KNL*. . Research Report RR-9035. Inria Bordeaux Sud-Ouest ; CNRS-IRIT ; Intel corporation ; Université Bordeaux, Feb. 2017, p. 15. URL: https://hal.inria.fr/hal-01473475.