Scalable Hybrid Programming and Performance for SuperLU Sparse Direct Solver

Sherry Li
Lawrence Berkeley National Laboratory

Piyush Sao **Rich Vuduc**
Georgia Institute of Technology

CUG 2014, May 4-8, 2014, Lugano, Switzerland

SuperLU_DIST: direct solver for general sparse linear systems on a distributed memory system

- first release in 1999
  - design target: each compute node with 1+ cores and UMA.
  - MPI only
- capable of factorizing matrices with millions of unknowns from real applications.
- available in Cray LibSci, PETSc, Trilinos, etc.
- used in many large-scale simulations, hybrid linear solvers, eigen solvers
  - 8000+ downloads in 2013.

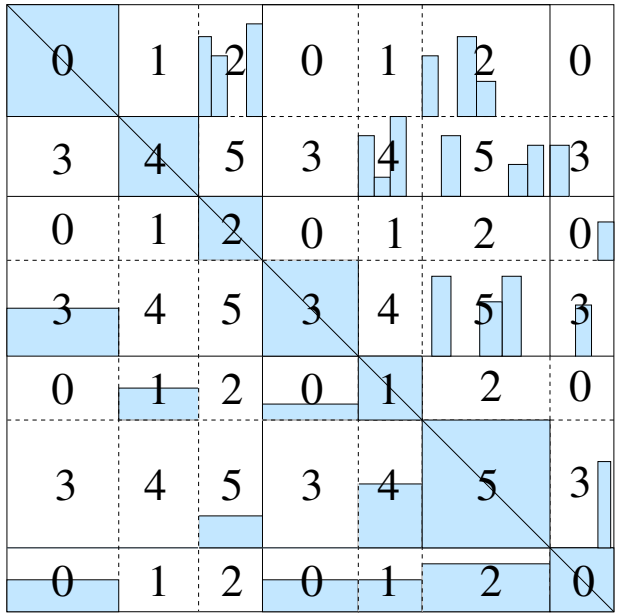
Outline

- Review of algorithms
- New intranode enhancements
  - aggregating small GEMM subproblems
  - pipelined execution
  - parallel Scatter with OpenMP
- Results

SuperLU_DIST: steps to solution

Compute factorization in three-stages:

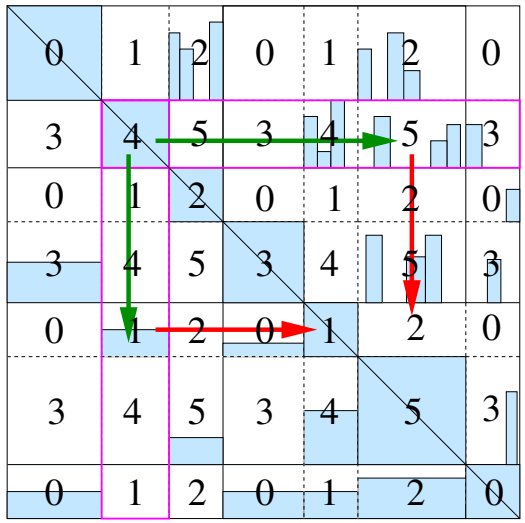
1. Matrix preprocessing:
   - static pivoting/scaling/permutation to improve numerical stability, to pereseve sparsity, to increase parallelism.
2. Symbolic factorization:
   - compute e-tree, structure of LU, static comm./comp. schedules
   - find supernodes (6-50 cols) for efficient dense block operations
3. Numerical factorization:
   - right-looking (fan-out, outer-product)
   - 2D cyclic MPI process grid



Compute solution with forward/back substitutions.

SuperLU_DIST: numerical factorization

right-looking factorization
for $j = 1, 2, \ldots, N_s$
  panel factorization (column and row)
    factor $A_{j,j} = L_{j,j}U_{j,j}$ and
      isend to $P_c(j)$ and $P_r(j)$
    If $p_{id} \in P_c(j)$ then
      wait for $U_{j,j}$ and
        factor $A_{(j+1):n_s,j}$, send to $P_r(:)$
    If $p_{id} \in P_r(j)$ then
      wait for $L_{j,j}$ and
        factor $A_{j,(j+1):n_s}$, send to $P_c(:)$
  Schur complement update
    update $A_{(j+1):n_s,(j+1):n_s}$
end for

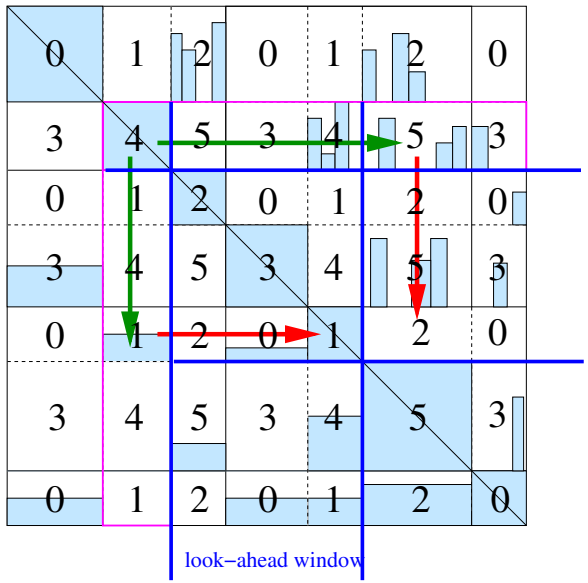


- panel factorization on critical path
- high parallelism, good load-balance for Schur complement update

Look-ahead in SuperLU_DIST with a fixed window size n_w

At each j-th step; factorize all "ready" panels in the window.
- reduce idle time of cores
- overlap comm. and comp.
- exploit more parallelism

for $j = 1, 2, \ldots, N_s$
  look ahead row factorization
  for $k = j+1, j+2, \ldots, j+n_w$ **do**
    if $L_{k,k}$ has arrived on $P_R(k)$ then
      factor $A_{k,(k+1):N_s}$ and isend to $P_C(:)$
  synchronizations
  wait for $L_{j,j}$ and factor $A_{j,j+1:N_s}$ if needed
  wait for $L_{:,j}$ and $U_{j,:}$
  look ahead column factorization
  for $k = j+1, j+2, \ldots, j+n_w$ **do**
    update $A_{:,k}$
    if possible then
      factor $A_{k:N_s,k}$ and isend to $P_R(:)$
  trailing matrix update
  update remaining $A_{(j+n_w+1):N_s,(j+n_w+1):N_s}$
end for



Challenges on manycores

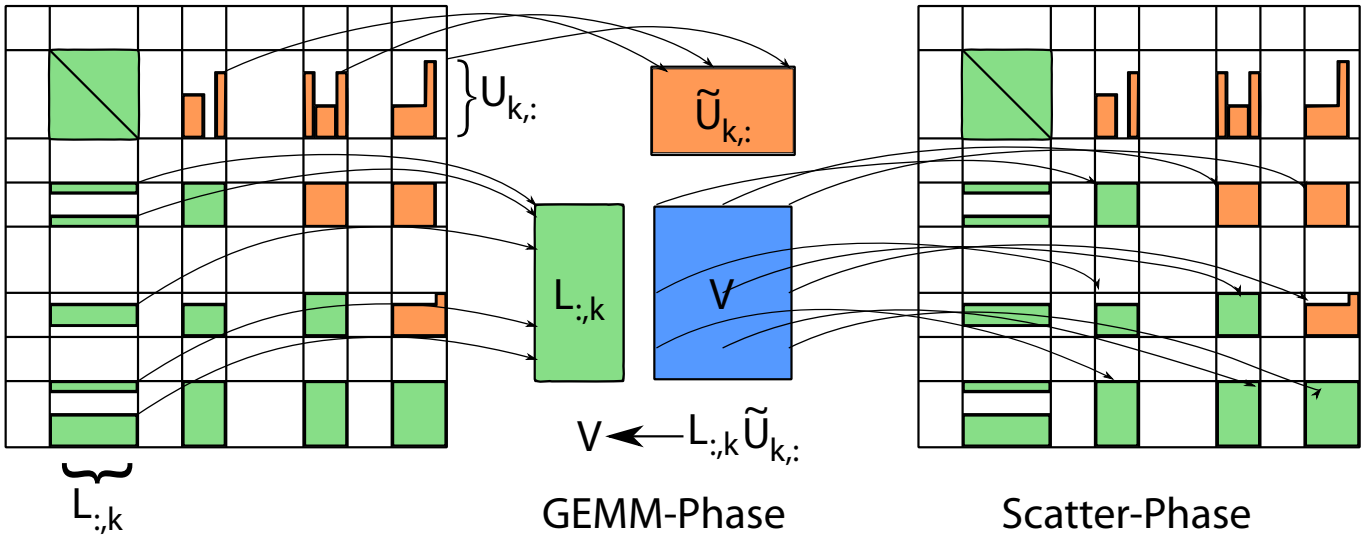
- neither strictly dominated by arithmetic, nor strictly dominated by communication
- indirect irregular memory access
- irregular parallelism
- strong dependence on the input matrix structure – known only at runtime

Goal: algorithm & data structure changes to use multithreading and GPU acceleration.

- MPI only $\Longrightarrow$ MPI+OpenMP+CUDA hybrid

1. CPU multithreading

- Abundant parallelism in Schur complement update
- Updates take 2 phases:
  1. GEMM phase: packing the U block, calling BLAS GEMM
  2. Scatter phase: unpacking the result into destination
- Optimization: aggregating small GEMM subproblems
  - enable use of multithreaded BLAS, and offloading GEMM to GPU
  - tradeoff: larger temp memory



Multithreading Scatter

- Insufficient just to use multithreaded BLAS (GEMM)
- Multithreading Scatter phase: How to assign blocks to threads?
  - static assignment leads to severe load imbalance
  - assigning one block to a thread is too fine grained, due to many small blocks
- Two strategies:
  - When enough number of columns, assign entire block column to a thread
  - when fewer block columns, parallelize across block rows

Test matrices

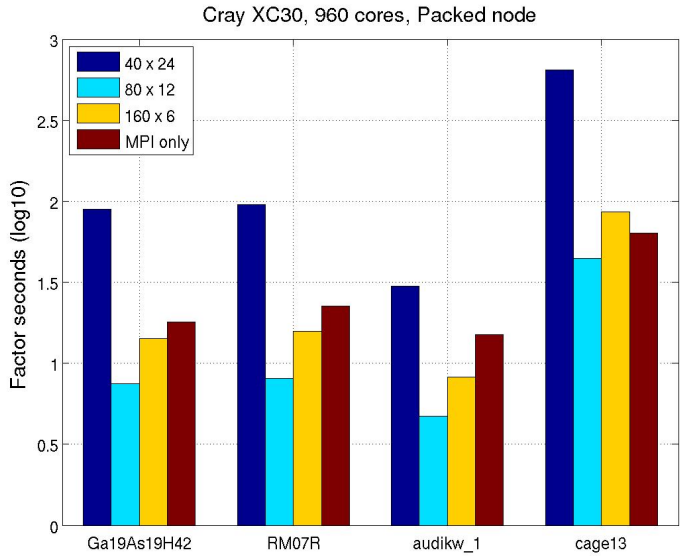
METIS ordering on $A + A^T$ to minimize fill-in

Name	n	nnz	$\frac{nnz}{n}$	symm	Fill Ratio	Application
Ga19As19H42	133123	8884839	67	yes	182	quantum chem.
RM07R	381689	37464962	98	no	78	CFD
audikw_1	943695	77651847	82	yes	31	structural
cage13	445315	7479343	17	no	55	DNA electrophoresis

Results on Cray XC30 "Cascade", edison at NERSC

- Intel Ivy Bridge processor @ 2.4 GHz
- 19.2 Gflops / core
- 2 x 12 cores per node
- 2.57 Petaflops/sec
- Cray Aries with Dragonfly topology, ~8 GB/sec MPI b.w.

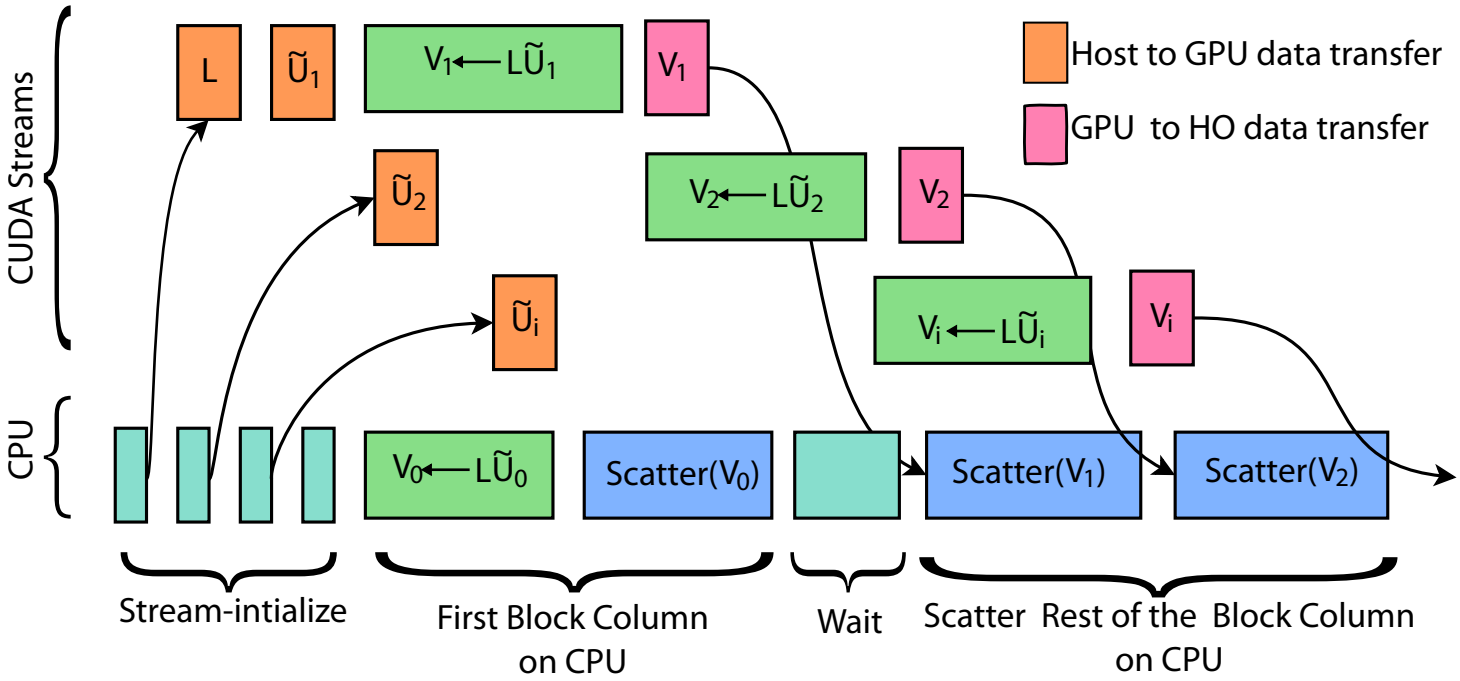
Fully packed mode, varying MPI-tasks x OMP-threads = 960 cores



2. GPU acceleration

Software pipelining to overlap CPU Scatter and GEMM copying to/from & execution on GPU

- divides $\tilde{U}$ into n_s partitions, n_s = number of desired CUDA streams (typical $n_s = 16$)
- GPU idle time is low



Different code configurations

Comparison baseline: SuperLU_DIST_3.3 latest release

- implicit parallelism
  - **MKL_p** : multithreaded MKL, 1 MPI per socket
  - {**cuBLAS,Scatter**} : MKL_p, only GEMM via cuBLAS
- explicit parallelism
  - **OpenMP+MKL_1** : single threaded MKL, OpenMP GEMM and Scatter
  - **OpenMP+{MKL_p,cuBLAS}** : CPU and GPU share GEMM
  - **OpenMP+{MKL_p,cuBLAS,Scatter}+pipeline** : CPU and GPU share GEMM

Jinx GPU cluster at Georgia Tech

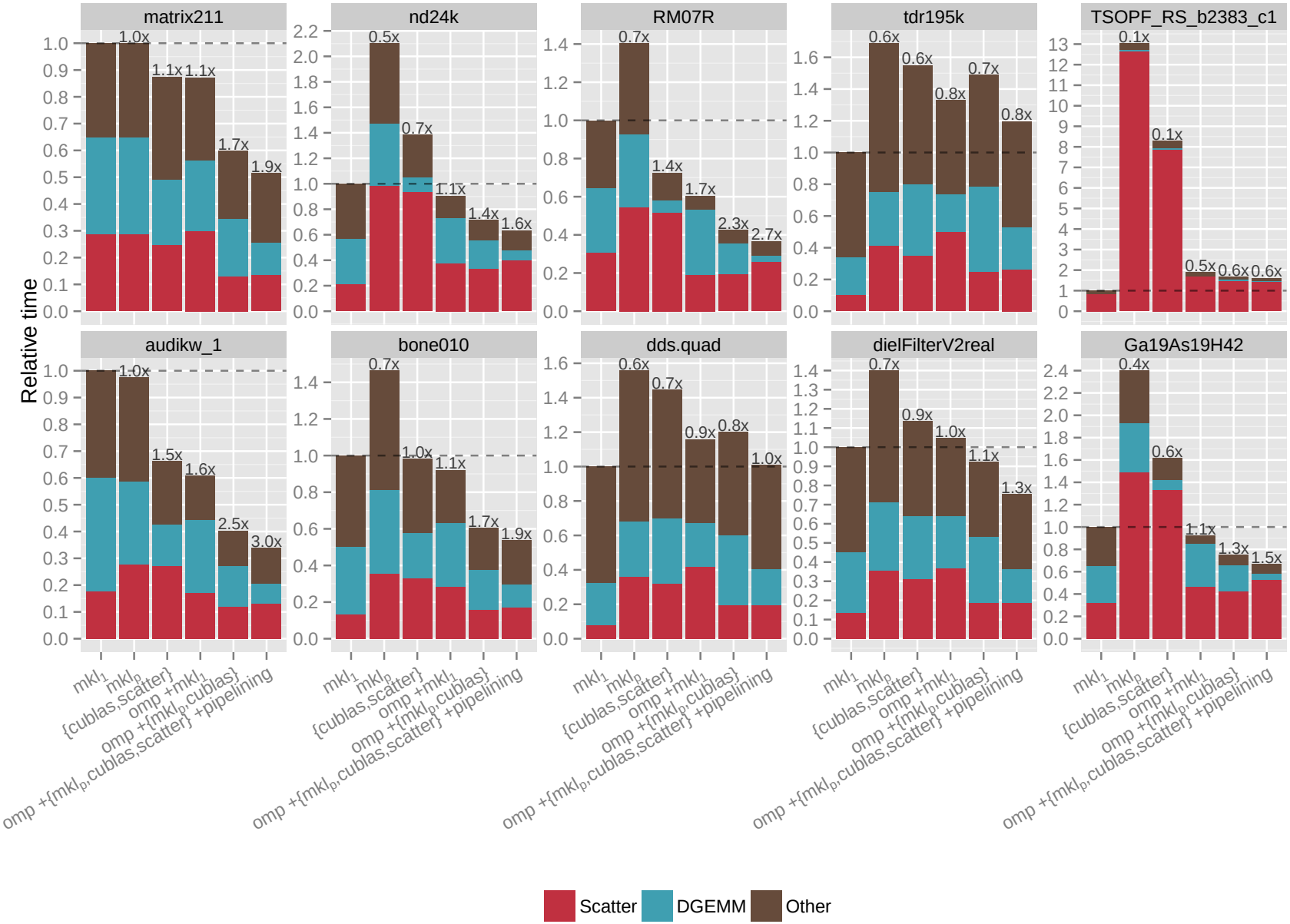
CPU

- Xeon X5550 @ 2.67 GHz
- 21.3 Gflops / core
- 2 x 6 cores per node
- InfiniBand, ∼10 GB/sec MPI bandwidth

GPU

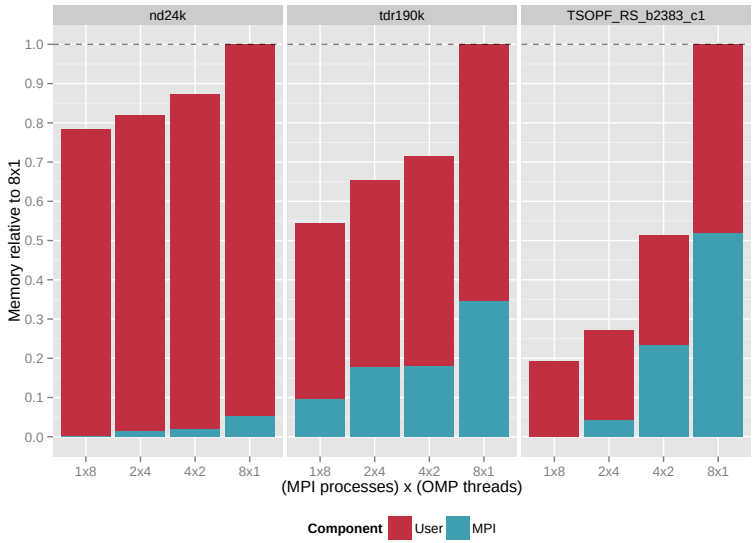
- Tesla M2090 “Fermi”
- 665 Gflops DP
-
- DRAM 6GB, 177 GB/sec bandwidth

Results on Jinx GPU cluster at Georgia Tech

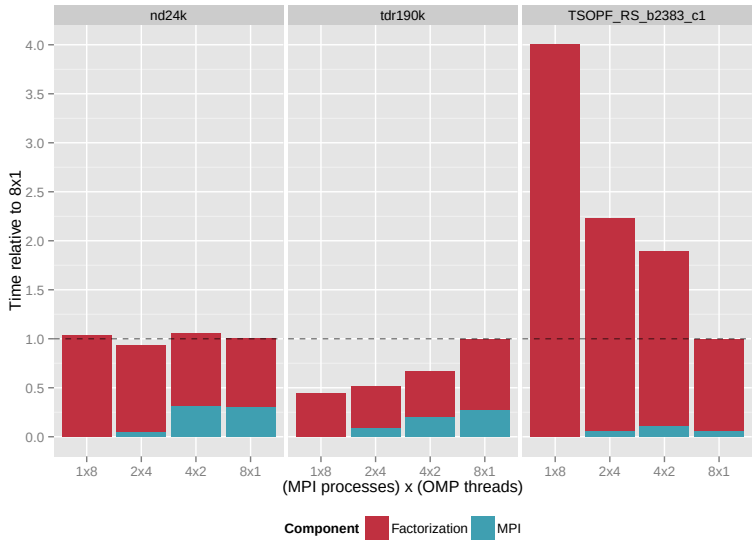


Effect of intranode threading on memory and time

3 test matrices



(a) User vs. MPI-runtime memory



(b) Time in MPI vs. compute

Summary

- implicit onnode parallelism is insufficient for sparse code
- MPI+OpenMP+CUDA delivers up to 3x time improvement, 2x memory saving

Summary

- implicit onnode parallelism is insufficient for sparse code
- MPI+OpenMP+CUDA delivers up to 3x time improvement, 2x memory saving

- larger GPU cluster: Titan, Blue Waters
- Intel Xeon Phi in progress
- OpenMP of the "other" part

Acknowledgment

This work was supported in part by the U.S. Dept. of Energy (DOE), Office of Science, Advanced Scientific Computing Research (and Basic Energy Sciences/Biological and Environmental Research/High Energy Physics/Fusion Energy Sciences/Nuclear Physics), through the Scientific Discovery through Advanced Computing (SciDAC) program; Additional support provided by the National Science Foundation (NSF) under NSF CAREER award number 0953100 and X-Stack 1.0 under DE-FC02-10ER26006/DE-SC0004915. We also used resources of the National Energy Research Scientific Computing Center, which is supported by the DOE Office of Science under Contract No. DE-AC02-05CH11231.