Exploiting Parallelism in Sparse Direct Solvers Part I

Xiaoye Sherry Li
xsli@lbl.gov
Lawrence Berkeley National Laboratory

Woudschoten Conference, Oct. 3-5, 2018

Plan

Focus on the following areas:

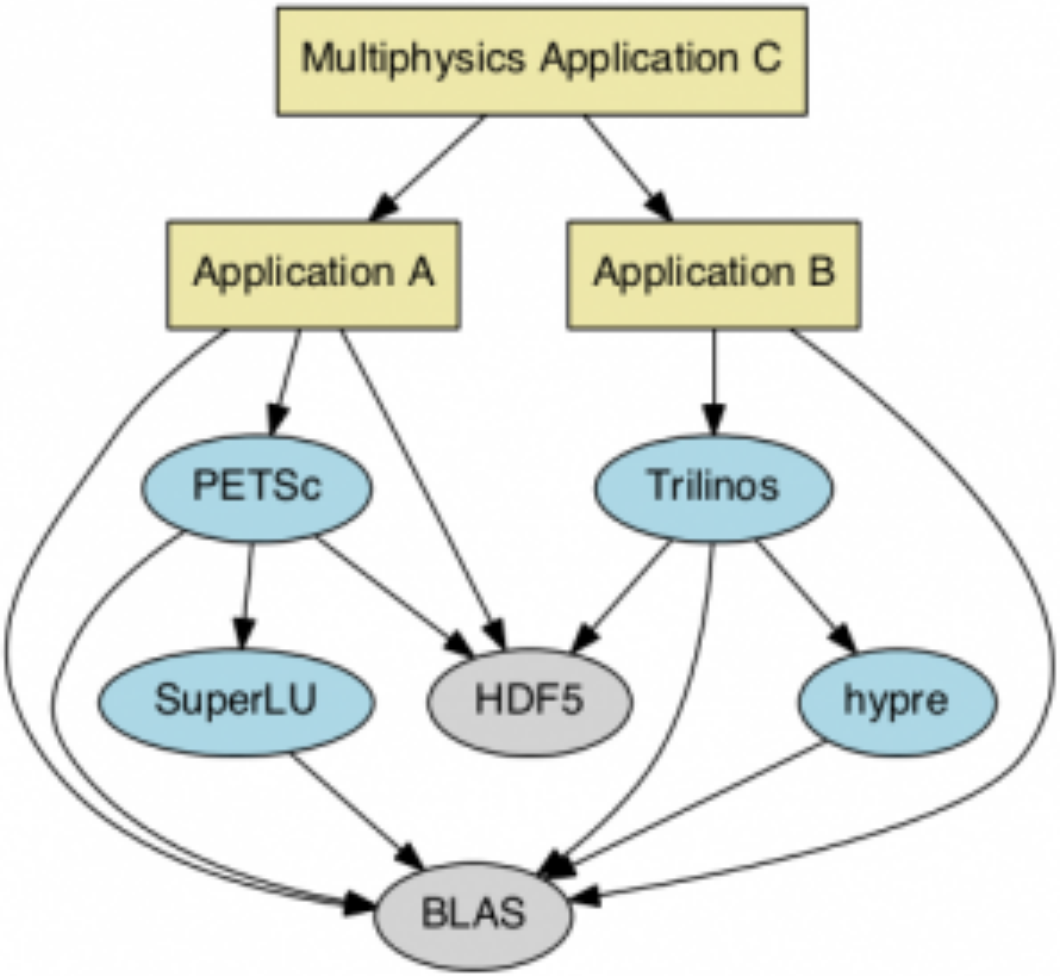
- Parallelism
- Numerical operations
- Implementation on HPC systems
- Analyzing performance bottlenecks

Sparse factorizations

- Core function for indefinite, ill-conditioned, algebraic equations (e.g., those from multiphysics, multiscale simulations)
  - **SuperLU**: direct solver
  - **STRUMPACK**: “inexact” direct solver, preconditioner
- Usage scenarios
  - Stand-alone solver
  - Good for multiple right-hand sides
  - Precondition Krylov solvers
  - Coarse-grid solver in multigrid (e.g., Hypre)
  - In nonlinear solver (e.g., SUNDIALS)
  - Solving interior eigenvalue problems
  - ….

➔ Bottom of the solvers toolchain. Can package as “black-box”

Example software stack



- Parallel machines
  - Shared memory
  - Shared address space
  - Message passing
  - Data parallel: vector processors
  - Clusters of SMPs
  - Cloud

- Programming model reflects hardware
  - Historically, tight coupling
  - Today, portability is important

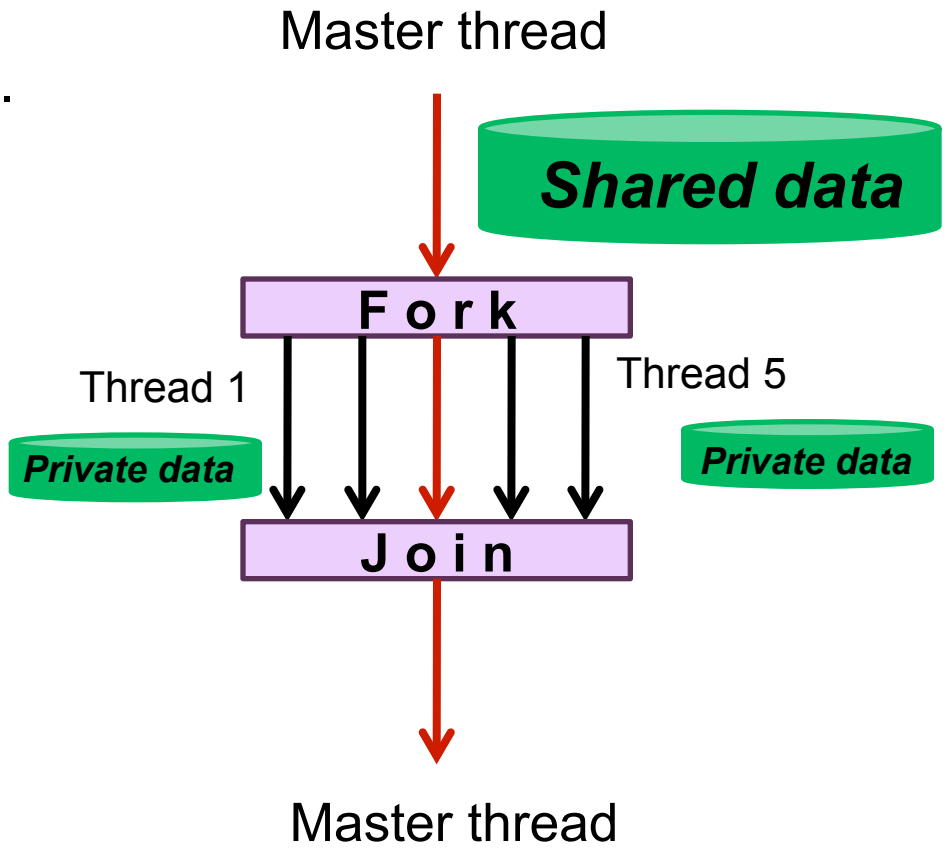
Parallel programming models

- Control
  - How is parallelism created?
  - What orderings exist between operations?
  - How do different threads of control synchronize?
- Data
  - What data is private vs. shared?
  - How is logically shared data accessed or communicated?
- Operations
  - What are the atomic (indivisible) operations?
- Cost
  - How do we account for the cost of each of the above?

OpenMP shared-memory programming

- Share the node address space.

- Most data shared within node.
- Threads communicate via memory read & write.
- Concurrent write to shared data needs *locking* or *atomic operation.*



Incorrect program

```
int s = 0;
```

```
Thread 1

  for i = 0, n/2-1
      s = s + x(i)*y(i)
```

```
Thread 2

  for i = n/2, n-1
      s = s + x(i)*y(i)
```

- There is a race condition on variable “s” in the program
- A race condition or data race occurs when:
  - two threads access the same variable, and at least one does a write.
  - the accesses are concurrent (not synchronized) so they could happen simultaneously

Correct program

```
int s = 0;
Lock lk;
```

Thread 1

```
local_s1 = 0
for i = 0, n/2-1
    local_s1 = local_s1 + x(i)*y(i)
lock(lk);
s = s + local_s1
unlock(lk);
```

Thread 2

```
local_s2 = 0
for i = n/2, n-1
    local_s2= local_s2 + x(i)*y(i)
lock(lk);
s = s +local_s2
unlock(lk);
```

- Most computation is on private variables
  - Sharing frequency is also reduced, which might improve speed
  - Race condition is fixed by adding locks to critical region (only one thread can hold a lock at a time; others wait for it)
- Shared-memory programming standards: OpenMP, Pthreads

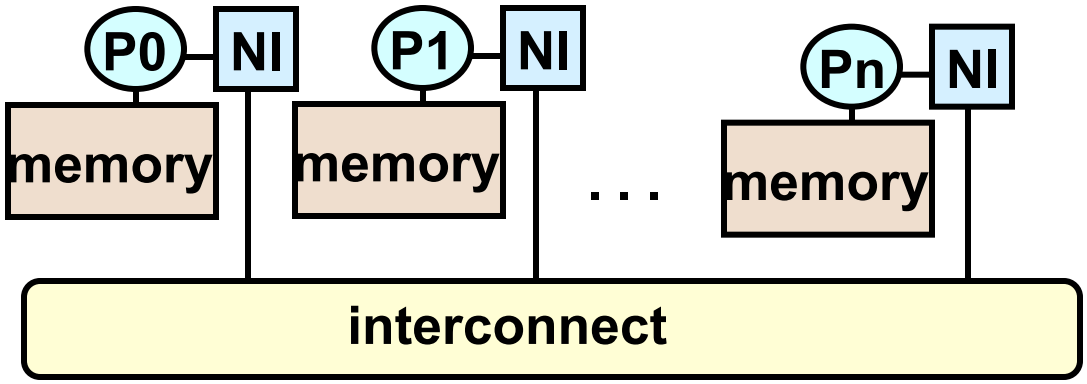
Dot product using OpenMP in C

```
int n = 100;
double x[100], y[100];
double s = 0, local_s;

#pragma omp parallel shared (s) private (local_s)
{
   local_s = 0.0;
   #pragma omp for
      for (i = 0; i < n; ++i) {
          local_s = local_s + x[i] * y[i];
      }
   #pragma omp critical
   {
       s = s + local_s;
   }
}
```

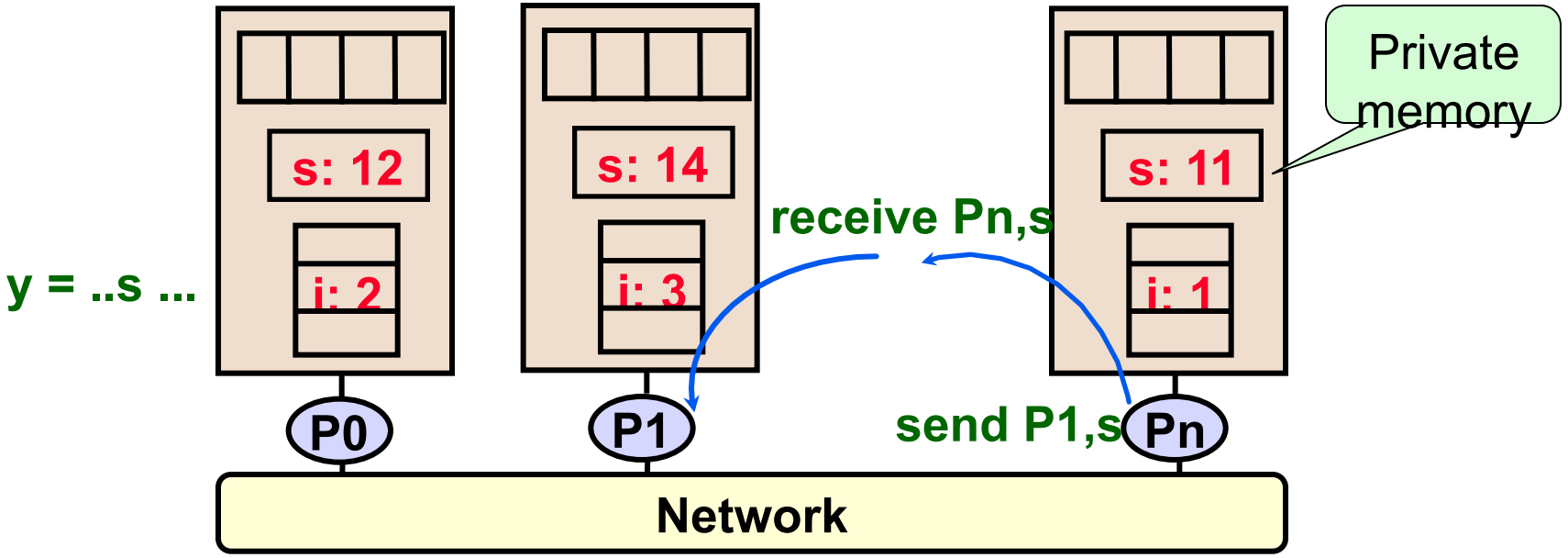
Machine model 2: distributed memory

- Each processor has its own memory and cache, but cannot directly access another processor's memory.
- Each "node" has a Network Interface (NI) for all communication and synchronization.



MPI distributed-memory programming

- Program consists of a collection of named processes
  - **Usually fixed at program startup time**
  - **Thread of control plus local address space – NO shared data**
- Processes communicate by explicit send/receive pairs
  - **Coordination is implicit in every communication event**



Distributed dot product

Processor 1

```
s = 0
for i = 0, n/2-1
      s = s + x(i)*y(i)
MPI_Recv(s_remote, p2,...)
MPI_Send(s, p2, ...)
s = s + s_remote
```

Processor 2

```
s = 0
for i = 0, n/2-1
      s = s + x(i)*y(i)
MPI_Send(s, p1, ...)
MPI_Recv(s_remote, p1,...)
s = s + s_remote
```

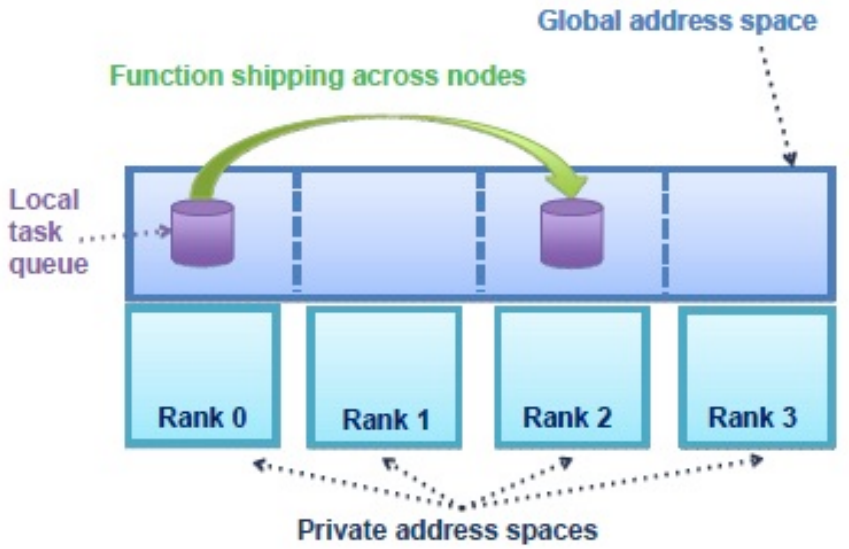
MPI – the de facto standard

- MPI has become the de facto standard for parallel computing using message passing
- Pros and Cons
  - MPI created a standard for applications development in the HPC community → portability
  - The MPI standard is a least common denominator building on mid-80s technology, so may discourage innovation

- MPI tutorial:
  https://computing.llnl.gov/tutorials/mpi/
  https://computing.llnl.gov/tutorials/mpi/exercise.html

Other machines & programming models

- Data parallel
  - SIMD / vector: Intel AVX2, AVX-512 (KNL)
  - GPU, at a larger scale
- Hybrid: cluster of multicore & GPU nodes
- MPI + X: (X = OpenMP, CUDA/OpenCL, …)
- Global Address Space programming (GAS languages)
  - UPC++, https://bitbucket.org/berkeleylab/upcxx
  - Local and shared data, as in shared memory model
  - But, shared data is partitioned over multiple processes
  - RMA access: Get / Put
  - Remote Procedure Call (RPC)



Amdahl's law bounding maximum speedup

- Suppose only part of an application is parallel
- Amdahl's law
  - Let s be the fraction of work done sequentially, so (1-s) is parallelizable
  - P = number of cores

$$\text{Speedup}(P) = \text{Time}(1)/\text{Time}(P)$$

$$<= 1/(s + (1-s)/P)$$

$$<= 1/s$$

(e.g., s = 1% → speedup <= 100)

- Even if the parallel part speeds up perfectly, performance is limited by the sequential part

Overheads of parallelism

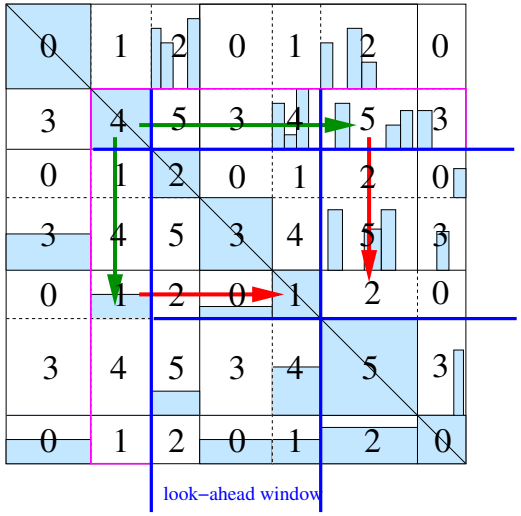
- Overheads include:
  - cost of starting a thread or process
  - cost of communicating shared data
  - cost of synchronizing
  - extra (redundant) computation

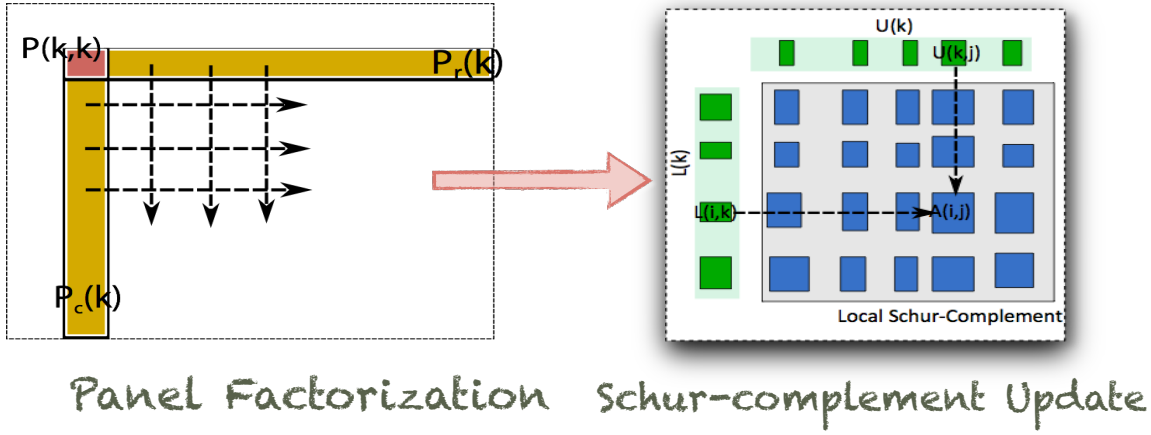
- Each of these can be in the range of milliseconds (= millions of flops) on some systems

- Tradeoff: Algorithm needs sufficiently large units of work to run fast in parallel (i.e. large granularity), but not so large that there is not enough parallel work

Sketch of SuperLU direct solver – communication pattern

XL, J. Demmel, J. Gilbert, L. Grigori, Y. Liu, P. Sao, Meiyue Shao, I. Yamazaki

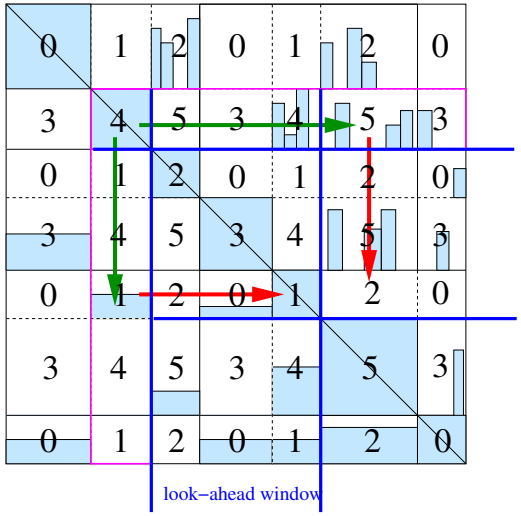


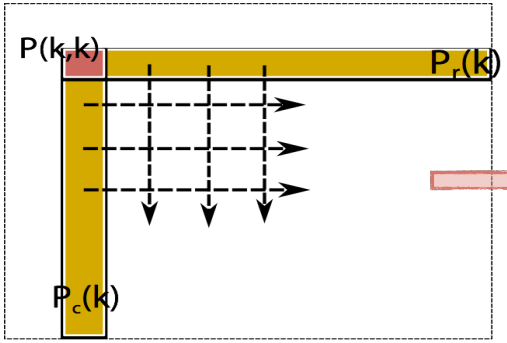


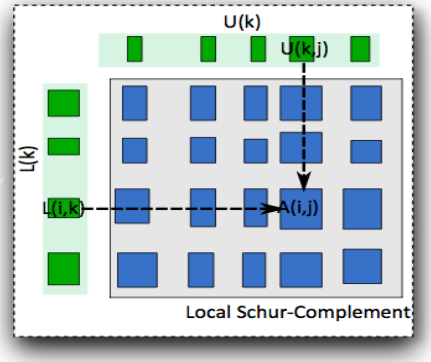
- Graph at step k+1 differs from step k
- Panel factorization on critical path

SuperLU direct solver – communication pattern

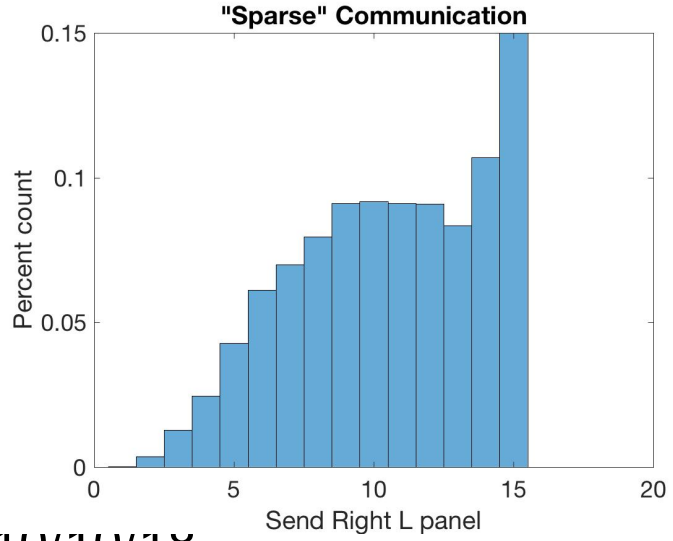
Developers: XL, J. Demmel, J. Gilbert, L. Grigori, P. Sao, Meiyue Shao, I. Yamazaki







Panel Factorization Schur-complement Update



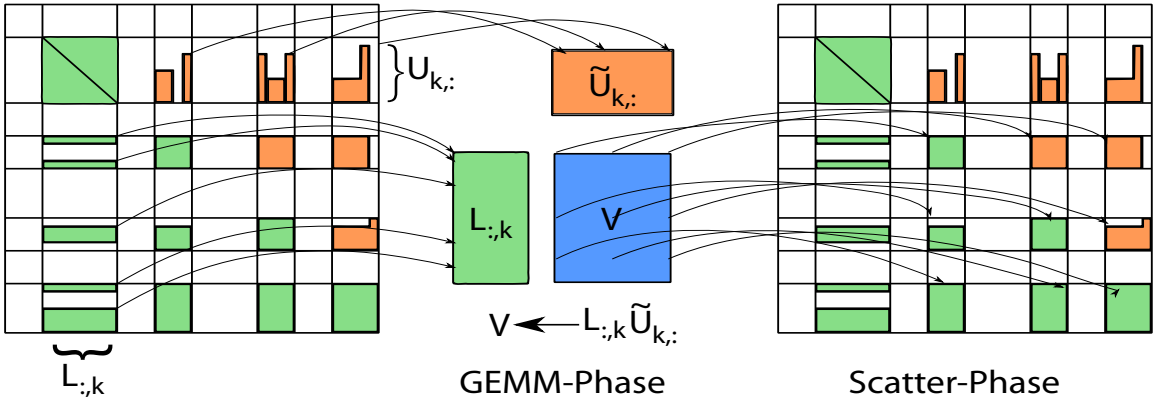
- Graph at step k+1 differs from step k
- Panel factorization on critical path

8 x 16 MPI process grid: Send-Right profile

Local computation

Schur complement update on each MPI rank

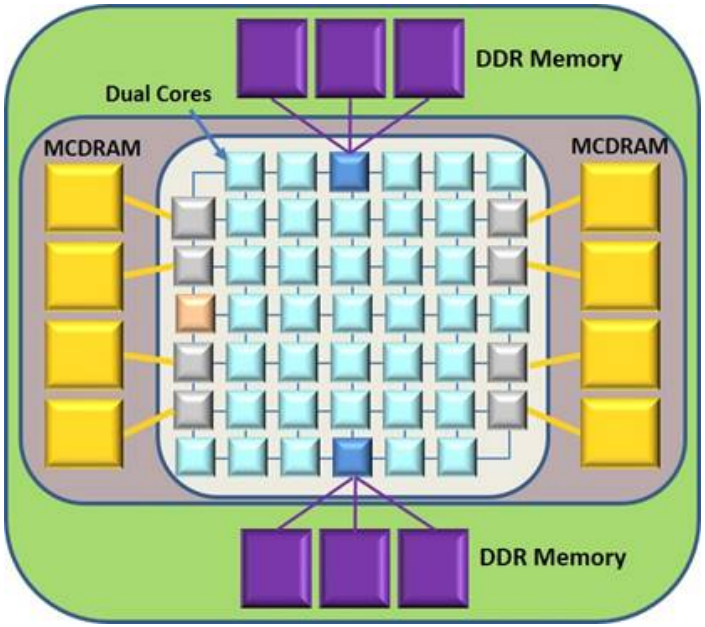
Loop through N steps: (Gaussian Elimination)
FOR (k = 1, N) {
1) **Gather** sparse blocks A(:, k) and A(k,:) into dense work[]
2) Call dense **GEMM** on work[]
3) **Scatter** work[] into remaining sparse blocks
}



Intel Xeon Phi: Knights Landing

Cray XC40 supercomputer at NERSC:

- 9688 KNL nodes: single socket
- 2388 Haswell nodes: 2 sockets X 16 cores



KNL node

- 72 cores @ 1.3 GHz, self hosted
- 2 cores form a tile
- 4 hardware threads per core (272 threads)
- 2 512-bit (8 doubles) vector units (SIMD)

Memory hierarchy

- L1 cache per core, 64 KB
- L2 cache per tile (2 cores share), 1MB
- 16 GB MCDRAM, >400 GB/s peak bandwidth
- 96 GB DDR4, 102 GB/s peak bandwidth

SuperLU optimization on Cori KNL node (1/2)

Work with Sam Williams, Jack Deslippe, Steve Leak, Thanh Phung

- Replace small independent single-threaded MKL GEMMs by large multithreaded MKL GEMMs: **15-20% faster.**
- Use new OpenMP features: **10-15% faster.**
  - "task parallel" to reduce load imbalance
  - "nested parallel for" to increase parallelism
- Vectorizing Gather/Scatter: **10-20% faster.**
  - **Hardware support: Load Vector Indexed / Store Vector Indexed**

```
#pragma omp simd     // vectorized Scatter
    for (i = 0; i < b; ++i) {
            nzval[ indirect2[i] ] = nzval[ indirect[i] ] - tempv[i];
    }
```

SuperLU optimization on Cori KNL node (2/2)

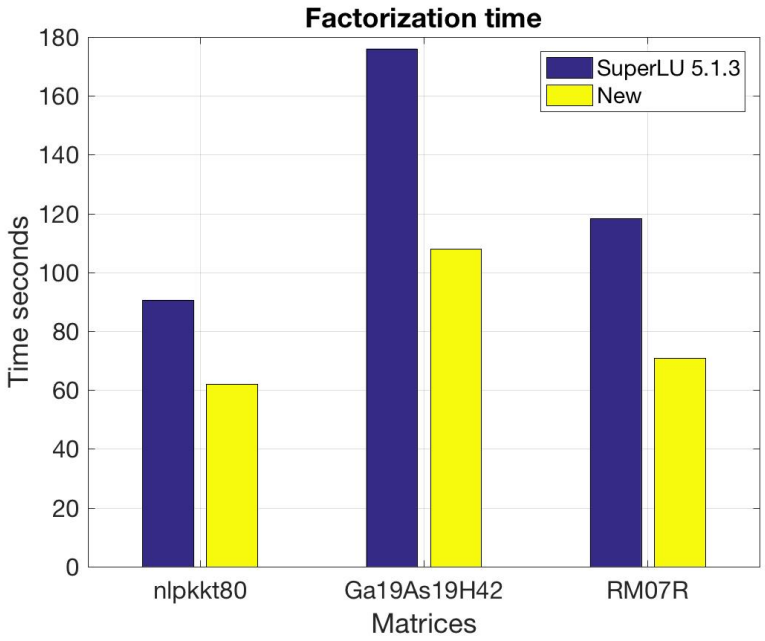
Reduce cache misses

- TLB (Translation Look-aside Buffer): a small cache for mapping virtual address to physical address
  - Large page requires smaller number of TLB entries

- Alignment during malloc
  - Page-aligned for large arrays → reduce TLB read frequency
  - CacheLine-aligned malloc for threads-shared data structures → reduce L1 read frequency, avoid false sharing

Single node improvement

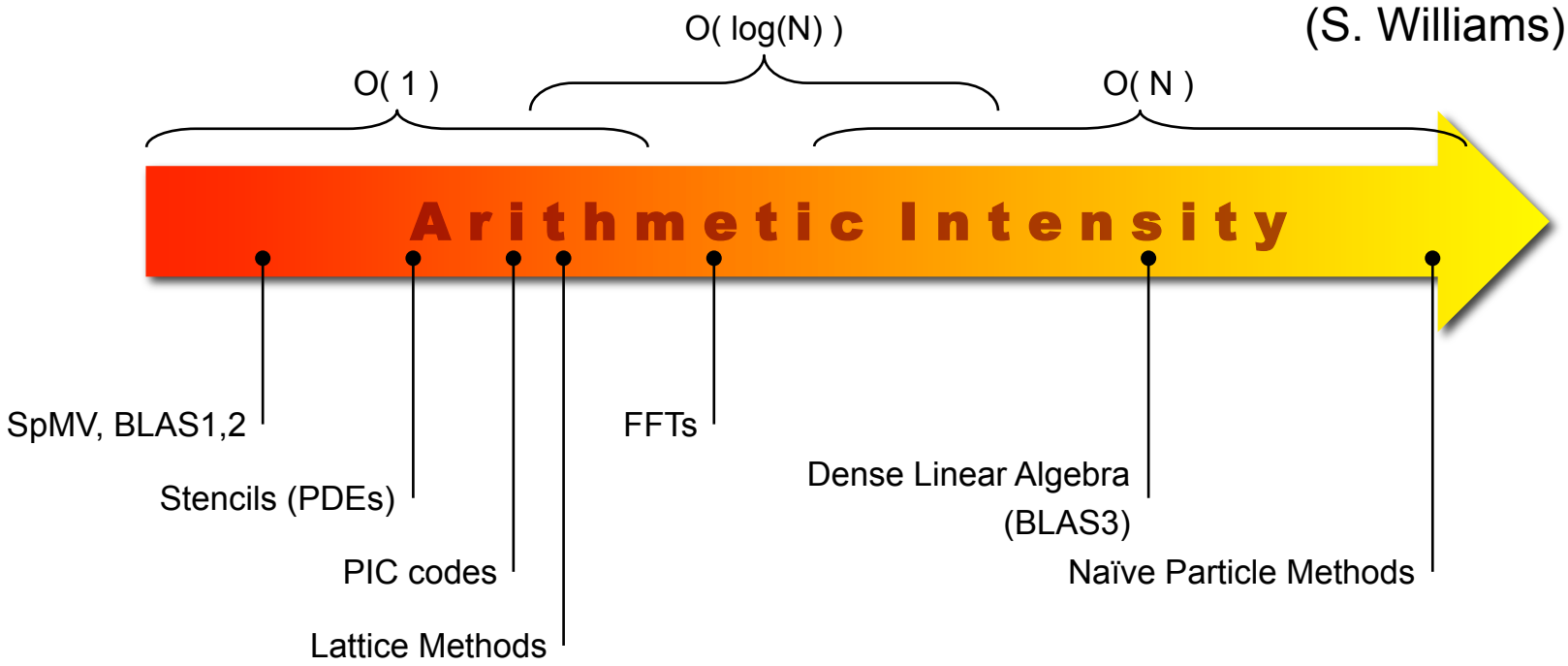
- up to 80% faster.
- Future work: auto generate/tune GEMM of different shapes.



nlpkkt80, n = 1.1M, nnz = 28M
Ga19As19H42, n = 1.3M, nnz = 8.8M
RM07R, n = 0.3M, nnz = 37.5M

Arithmetic Intensity

- Arithmetic Intensity (AI) ~ Total Flops / Total DRAM Bytes
  - E.g.: dense matrix-matrix multiplication: n^3 flops / n^2 memory
- Higher AI → better locality → amenable to many optimizations → achieve higher % machine peak



Roofline model (S. Williams)

- Is the code computation-bound or memory-bound?
- Synthesize communication, computation, and locality into a single visually-intuitive performance figure using bound analysis
  - Assume perfect overlap computation and communication w/ DRAM
  - Arithmetic Intensity (AI) is computed based on DRAM traffic

E.g.: DGEMM AI = 2*M*N*K / (M*K + K*N + 2*M*N) / 8

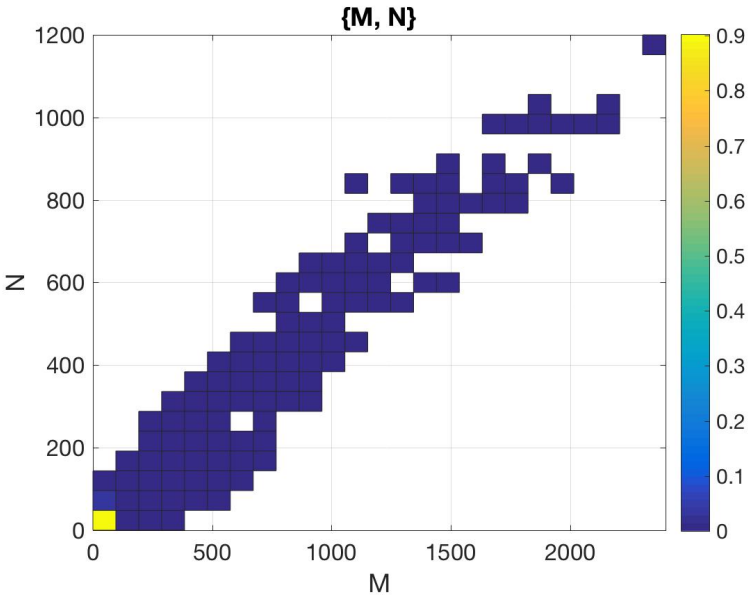
- Time is the maximum of the time required to transfer the data and the time required to perform the floating point operations.

$$\text{Attainable GFLOP/s} = \min \begin{cases} \text{Peak GFLOP/s} \\ \text{AI} * \text{Peak GB/s} \end{cases}$$

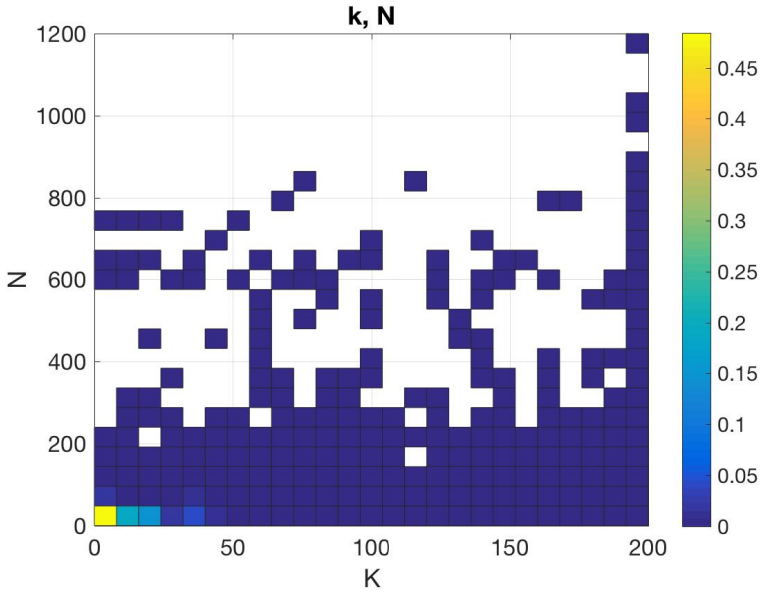
GEMM dimension profile

- nlpkkt80 : n = 1.1M, nnz = 28M, ~70,000 supernodes
- non-uniform block size, non-square, many blocks are small

GEMM {m, n} dimensions

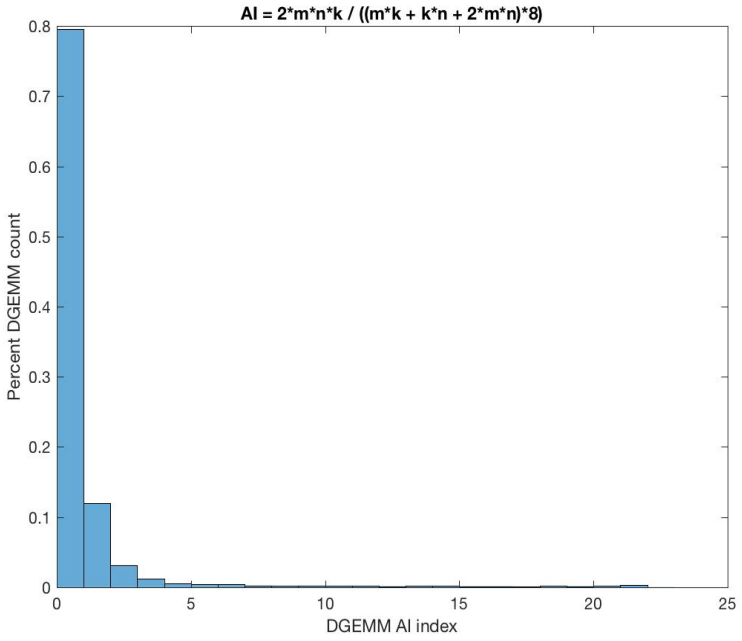


GEMM {k, n} dimensions

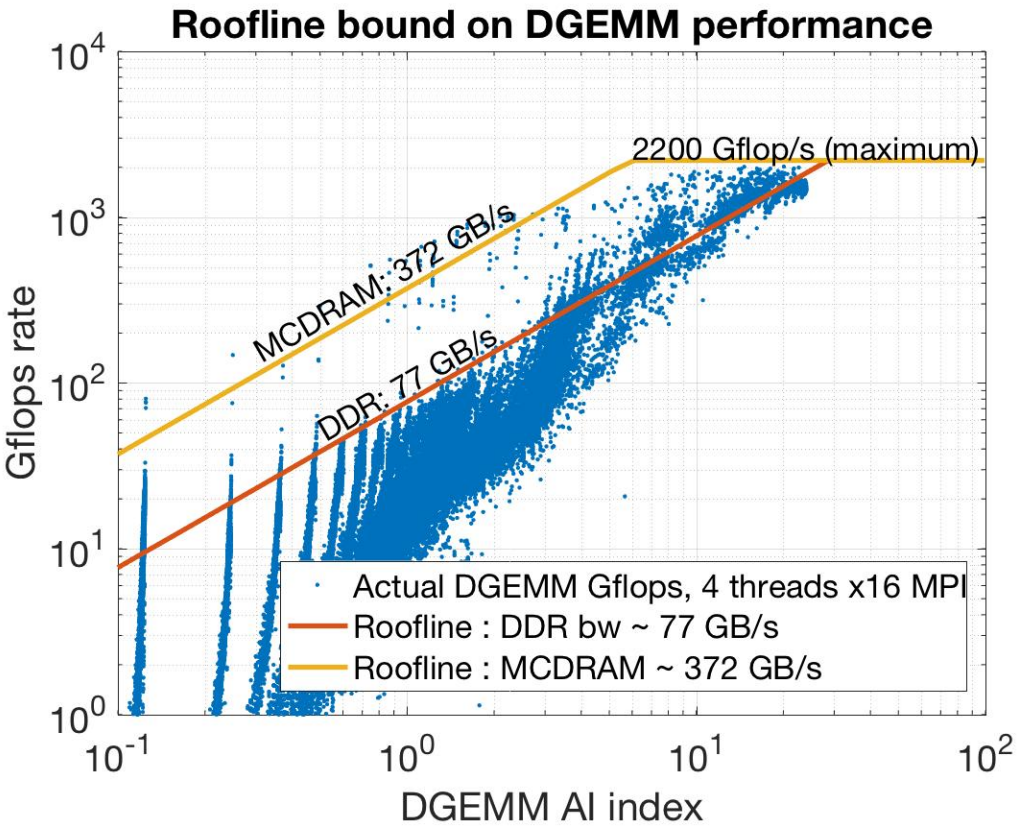


DGEMM roofline performance bound

AI = 2*M*N*K / (M*K + K*N + 2*M*N) / 8



DGEMMs performance profile



Tools for constructing Roofline model

- Software:

http://crd.lbl.gov/departments/computer-science/PAR/research/roofline

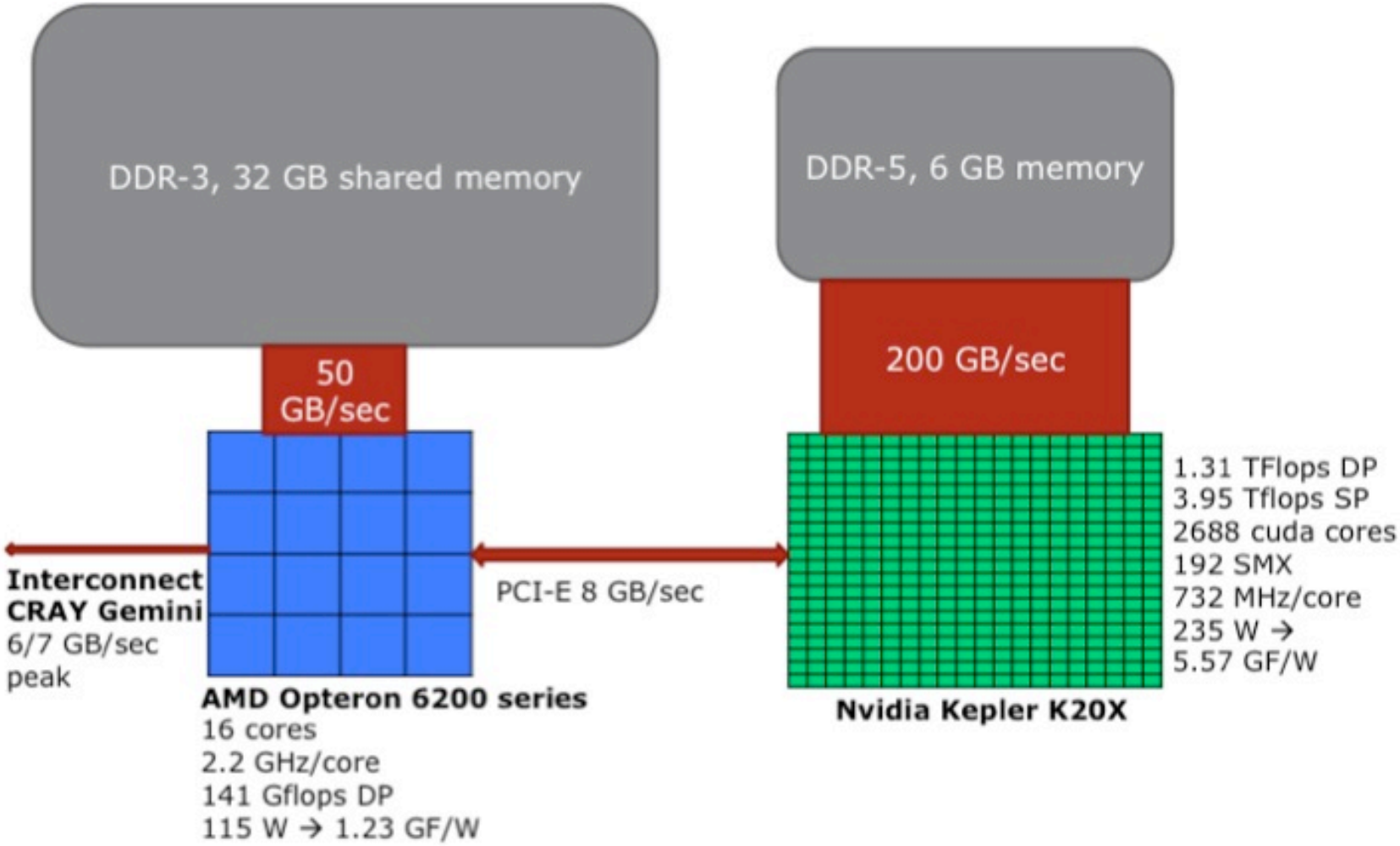
  - LIKWID: Erlangen Regional Computing Centre
    - Lightweight, support Intel, IBM POWER, ARM
  - SDEVTune: Intel
  - Advisor: Intel
  - NVProf: Nvidia

- Publications

http://crd.lbl.gov/departments/computer-science/PAR/research/roofline/publications/

GPU computing

Cray XK7 (Titan at ORNL): 16-core AMD + K20X GPU



Design questions for accelerator / co-processor

- Use CPU as well ?
  - current accelerator DRAM still small

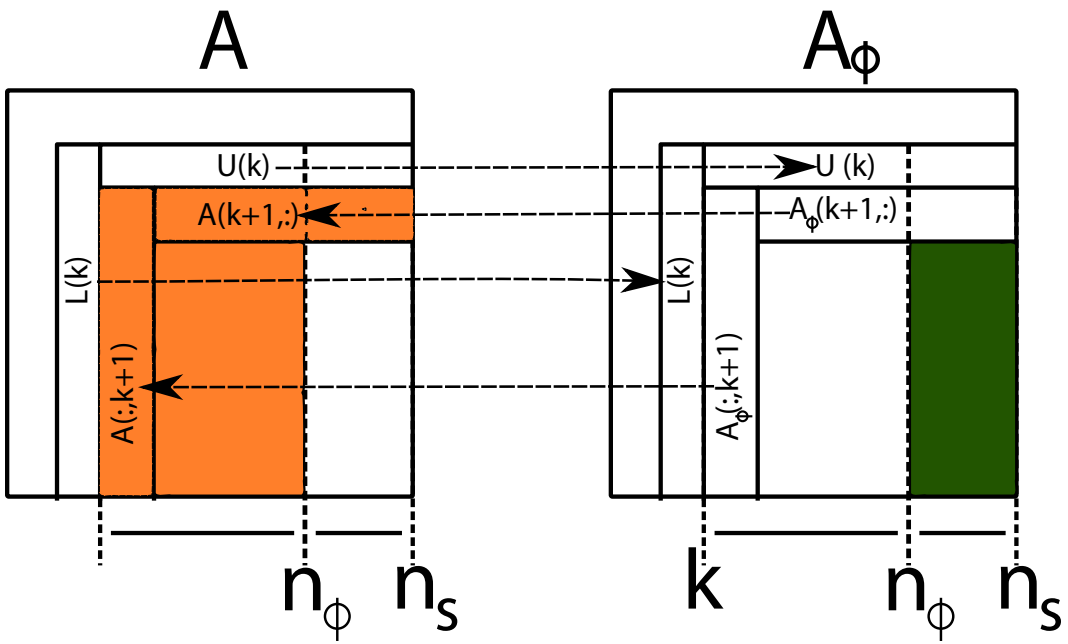
  → use “offload” mode.
- What to offload?
  - Panel factorization not suitable for fine-grained data-parallel model

  → offload only Schur complement update
- Schur complement update: GEMM, and Gather/Scatter?
  - GEMM only – compute intensive
  - Both GEMM and Gather/Scatter – indirect addressing, memory intensive.

- Key: overlap activities on both sides to hide PCIe latency

HALO algorithm – Highly Asynchronous Lazy Offload

- Two partial sums of Schur-complement are maintained separately on CPU and GPU
- Reduce to-be-factorized panel on CPU, absorbing GPU's panel

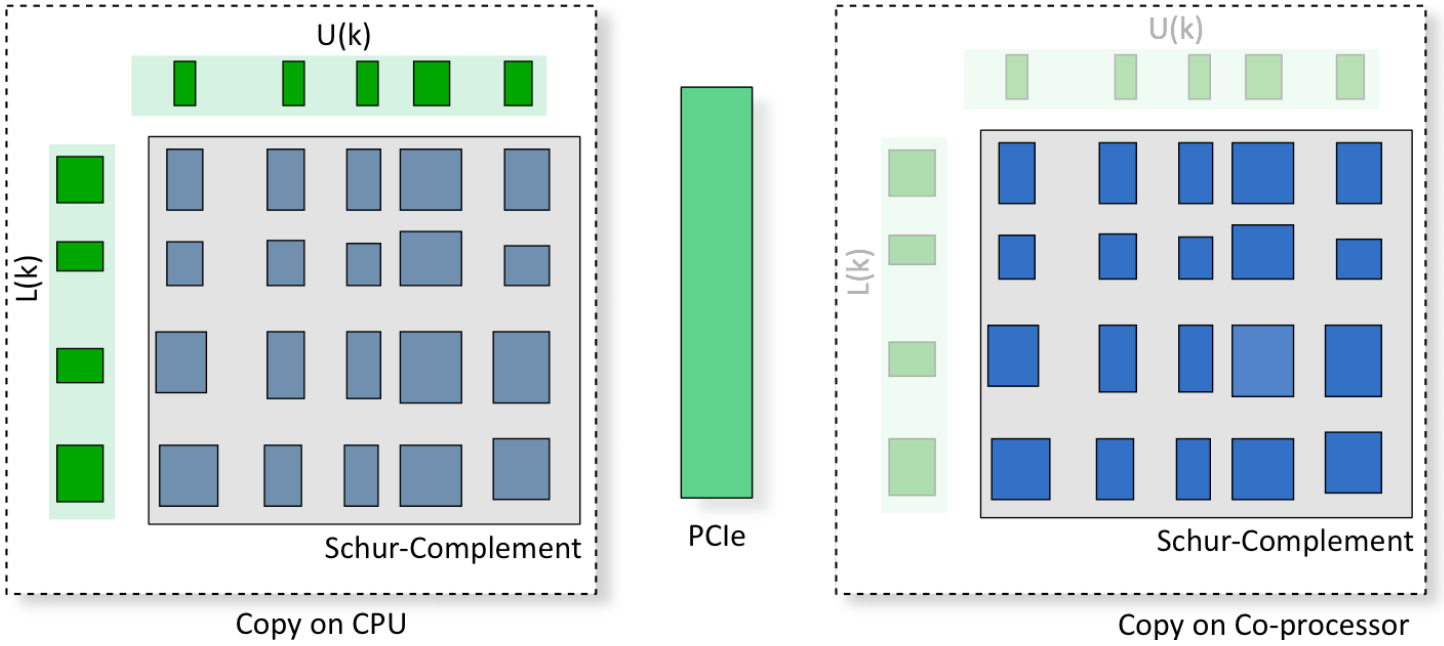


P. Sao, R. Vuduc, and X.S. Li, "A distributed CPU-GPU sparse direct solver", Proc. of Euro-Par 2014 Parallel Processing, August 25-29, Porto, Portugal.

P. Sao, X. Liu, R. Vuduc, and X.S. Li, "A Sparse Direct Solver for Distributed Memory Xeon Phi-accelerated Systems", IPDPS 2015, May 25-29, 2015, Hyderabad, India.

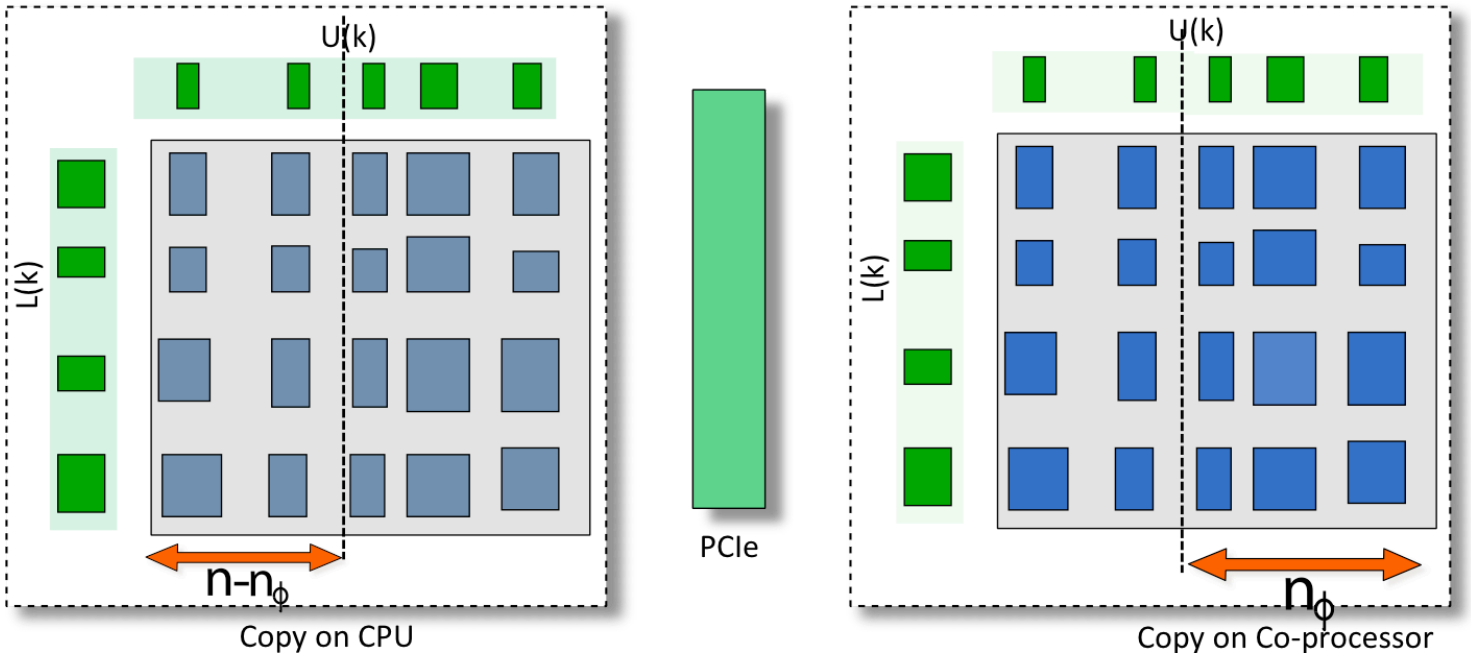
HALO algorithm – Highly Asynchronous Lazy Offload

- Two partial sums of Schur-complement are maintained separately on CPU and GPU
- Reduce to-be-factorized panel on CPU, absorbing GPU's panel



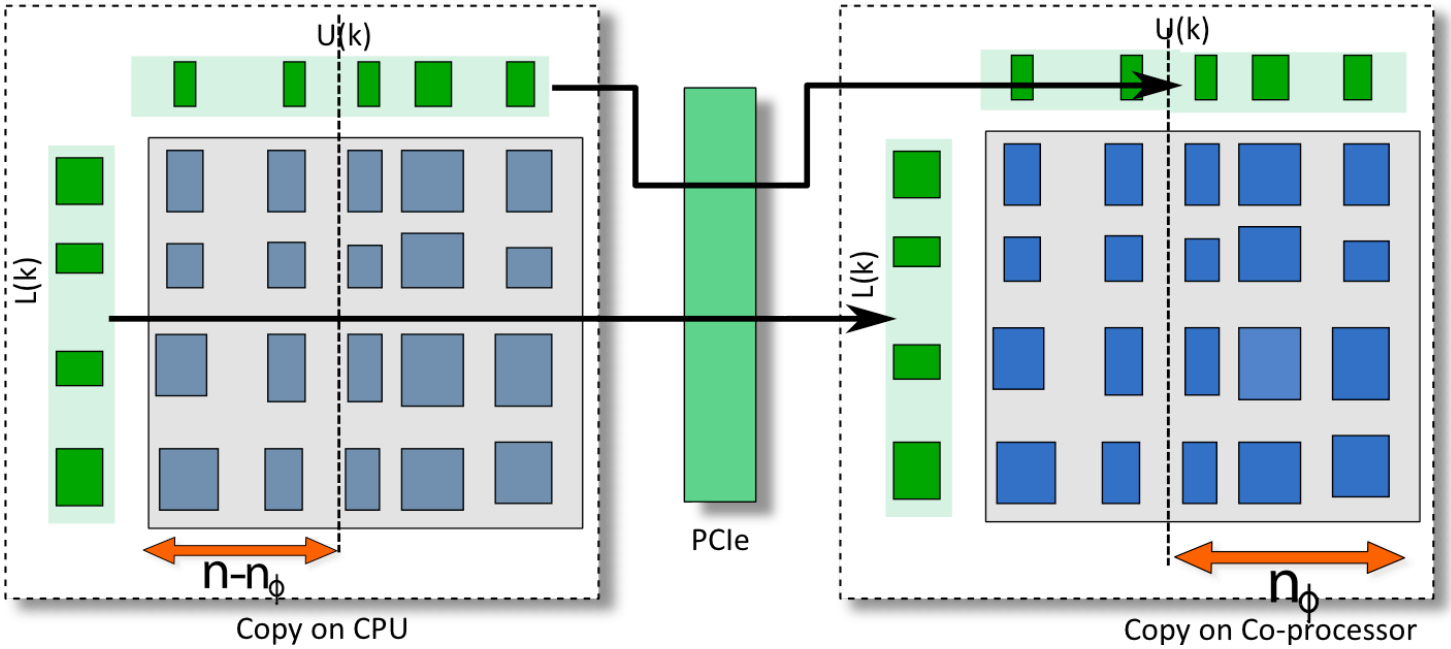
HALO algorithm – Highly Asynchronous Lazy Offload

- Two partial sums of Schur-complement are maintained separately on CPU and GPU
- Reduce to-be-factorized panel on CPU, absorbing GPU's panel



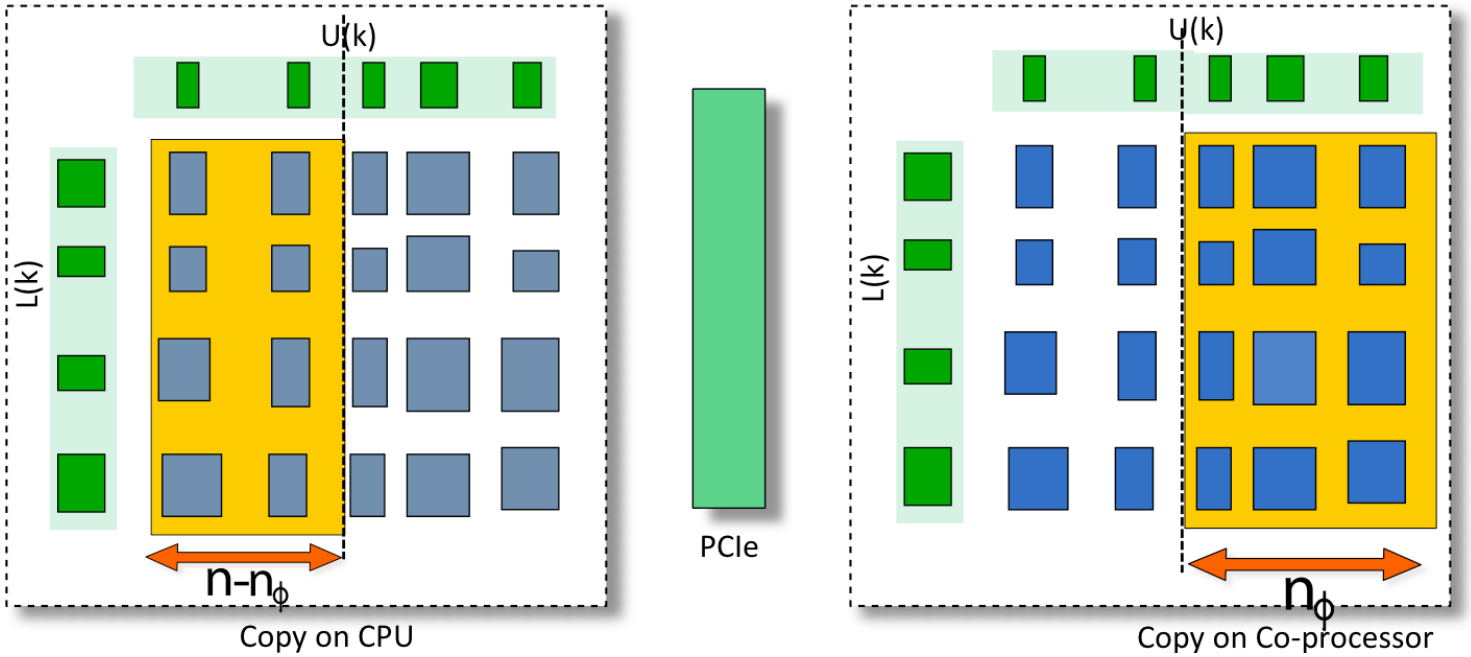
HALO algorithm – Highly Asynchronous Lazy Offload

- Two partial sums of Schur-complement are maintained separately on CPU and GPU
- Reduce to-be-factorized panel on CPU, absorbing GPU's panel



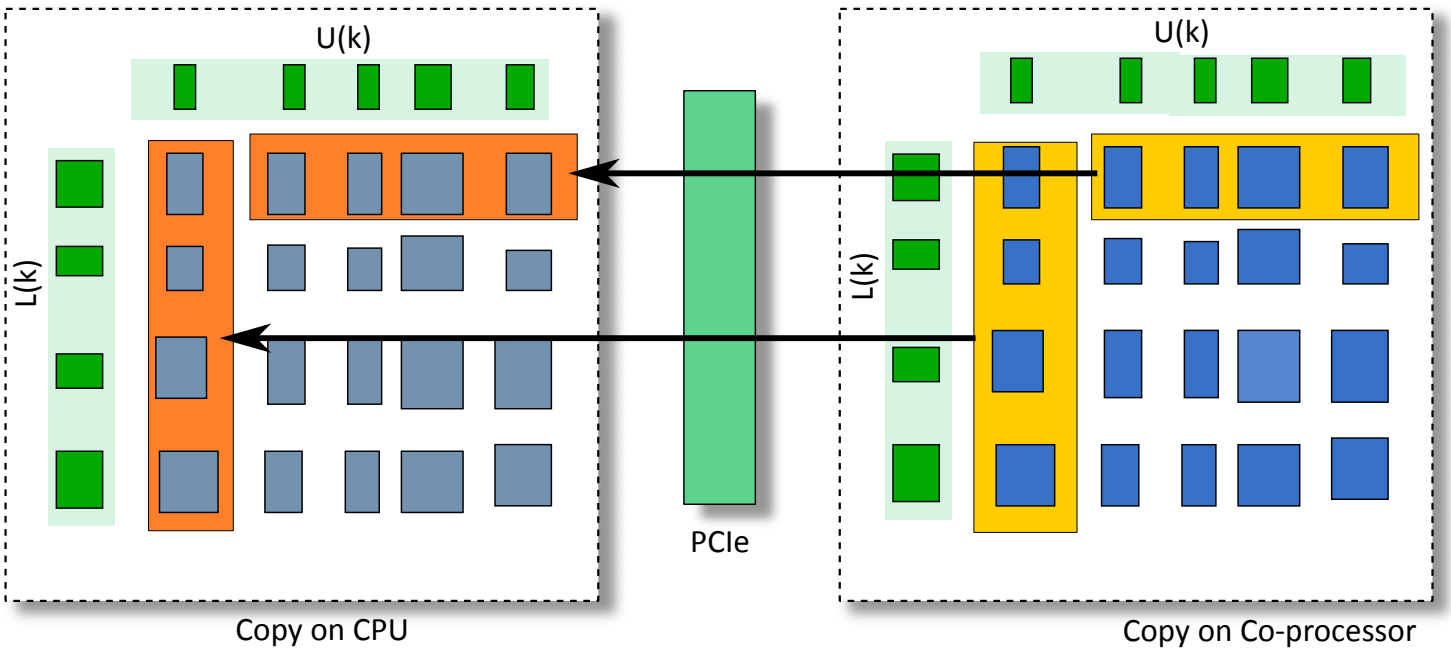
HALO algorithm – Highly Asynchronous Lazy Offload

- Two partial sums of Schur-complement are maintained separately on CPU and GPU
- Reduce to-be-factorized panel on CPU, absorbing GPU's panel

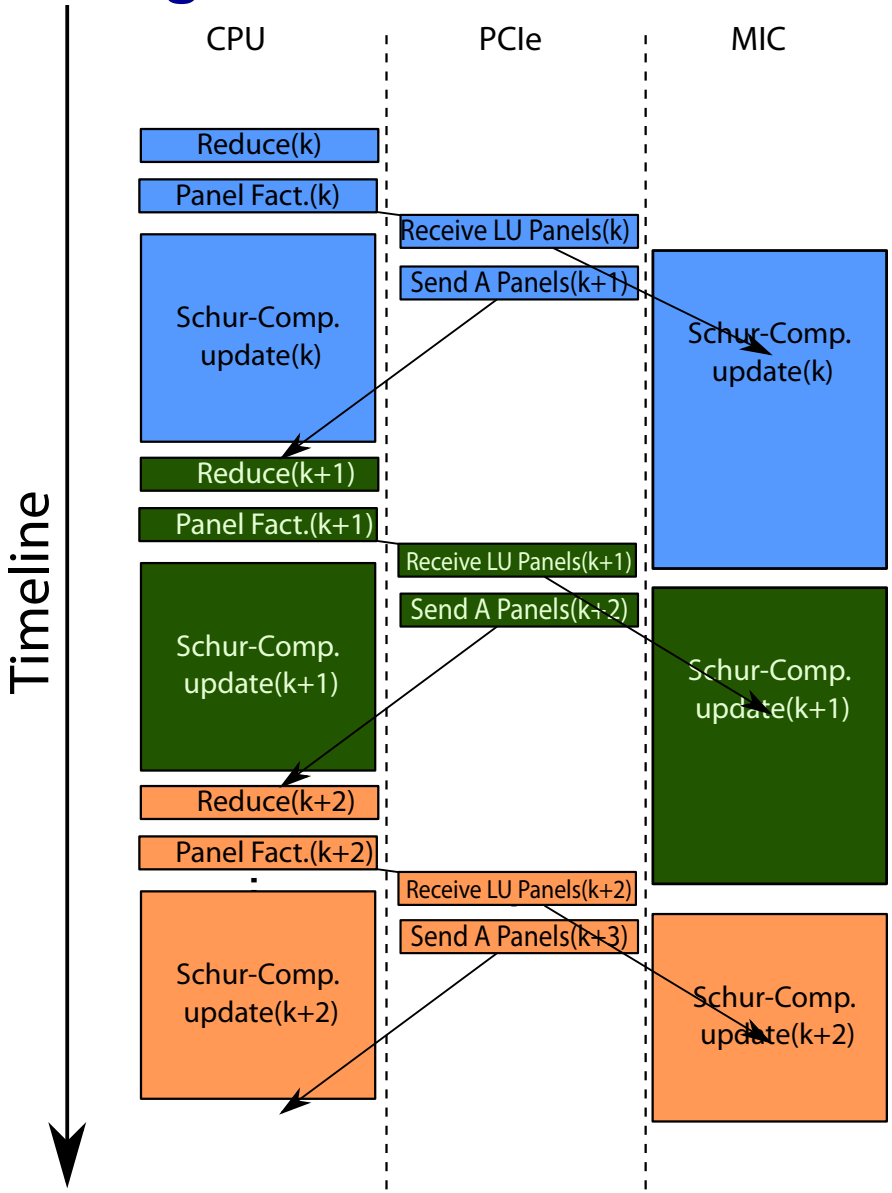


HALO algorithm – Highly Asynchronous Lazy Offload

- Two partial sums of Schur-complement are maintained separately on CPU and GPU
- Reduce to-be-factorized panel on CPU, absorbing GPU's panel



Software pipelining to hide PCIe costs



Programming in CUDA

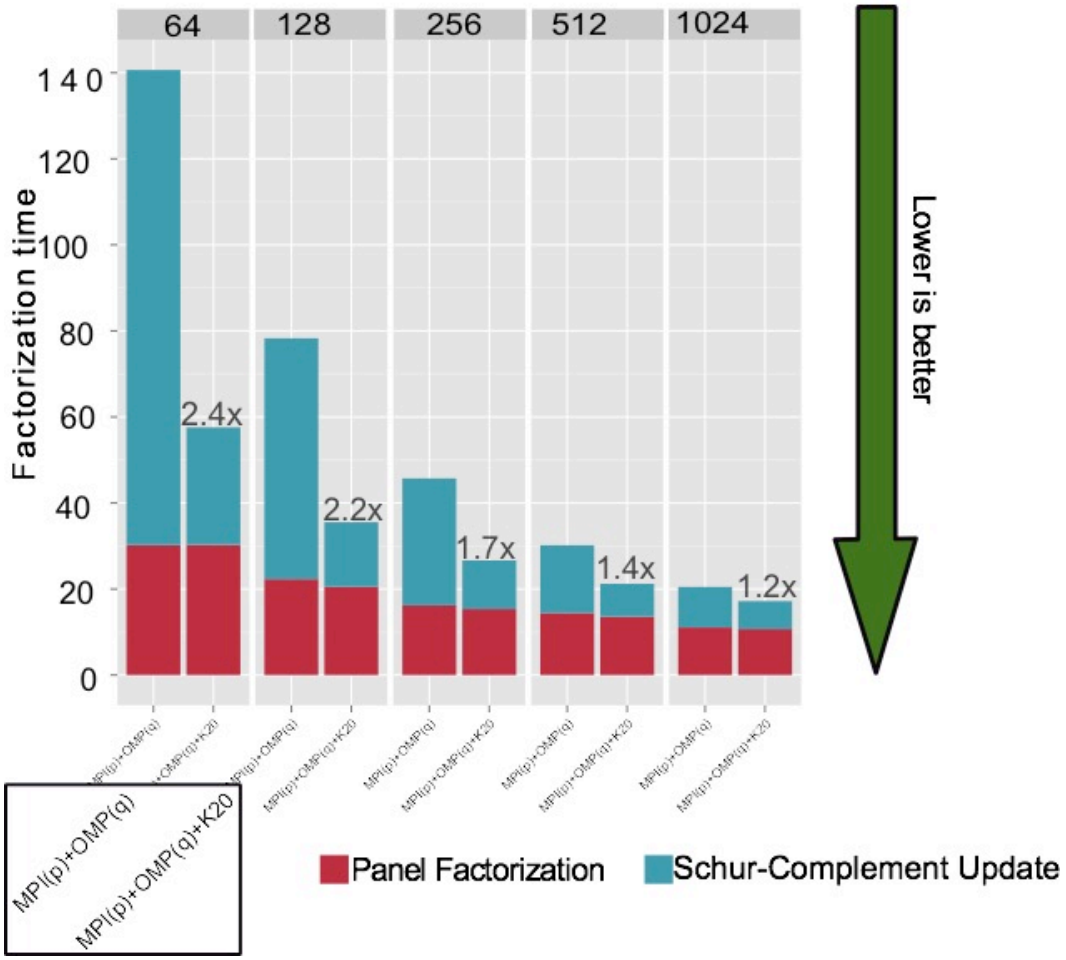
CPU and GPU have separate memory ... like distributed memory programming

At each step:

- For each CUDA stream:
  - cudaMemcpyAsync(…, HostToDevice)
  - cublasDgemm(); Scatter
  - cudaMemcpyAsync(…, DeviceToHost)

- CPU performs its own panel factorization, and Schur complement update

Strong scaling on Titan: MPI + OpenMP + CUDA



At large number of MPI processes, panel factorization becomes bottleneck

Summary

- Refactor existing codes and implement new codes for current and next-generation machines (exascale in 2021)
  - Fully exploit manycore node architectures
    - Vectorization, multithreading, …
    - GPU accelerator
  - Reduce communication and synchronization

- Explore new algorithms that require lower arithmetic complexity, communication and synchronization, faster convergence rate
  - SuperLU: new 3D algorithm to reduce communication
  - STRUMPACK: “inexact” direct solver, preconditioner, based on hierarchical low rank structures: HSS, HODLR, etc.

References

- Short course, “Factorization-based sparse solvers and preconditioners”, 4th Gene Golub SIAM Summer School, 2013.https://archive.siam.org/students/g2s3/2013/index.html
  - 10 hours lectures, hands-on exercises
  - Extended summary: http://crd-legacy.lbl.gov/~xiaoye/g2s3-summary.pdf

  (in book “Matrix Functions and Matrix Equations”, https://doi.org/10.1142/9590)
- “The Landscape of Parallel Processing Research: The View from Berkeley”

  http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/ECS-2006-183.pdf
  - Contains many references
- Jim Demmel, Kathy Yelick, et al., UCB/CS267 lecture notes for parallel computing class https://sites.google.com/lbl.gov/cs267-spr2018/

THANK YOU