SAND2016-3315C

Exceptional service in the national interest

Sandia National Laboratories

Task and Data Parallelism Based Direct Solvers and Preconditioners in Manycore Architectures: Efforts in Trilinos/ShyLU

Joshua Dennis Booth
jdbooth@sandia.gov
SIAM-PP 2016 (MS36)

Project Head: Siva Rajamanickam
Joint Work: Mehmet Deveci, Kyungjoo Kim, Andrew Bradley, Erik Boman
Collaborators: Clark Dohrmann, Heidi Thornquist

U.S. DEPARTMENT OF ENERGY | NNSA National Nuclear Security Administration | CCR Center for Computing Research

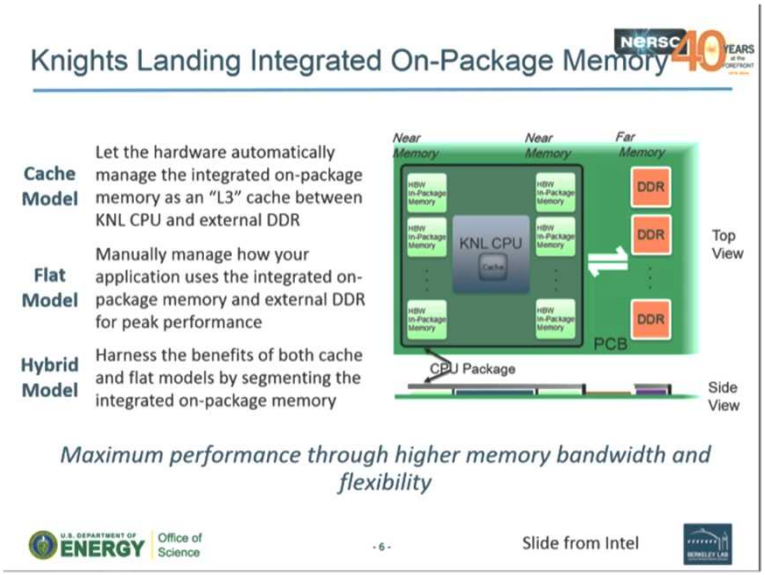
Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Need for thread-scalable linear solvers

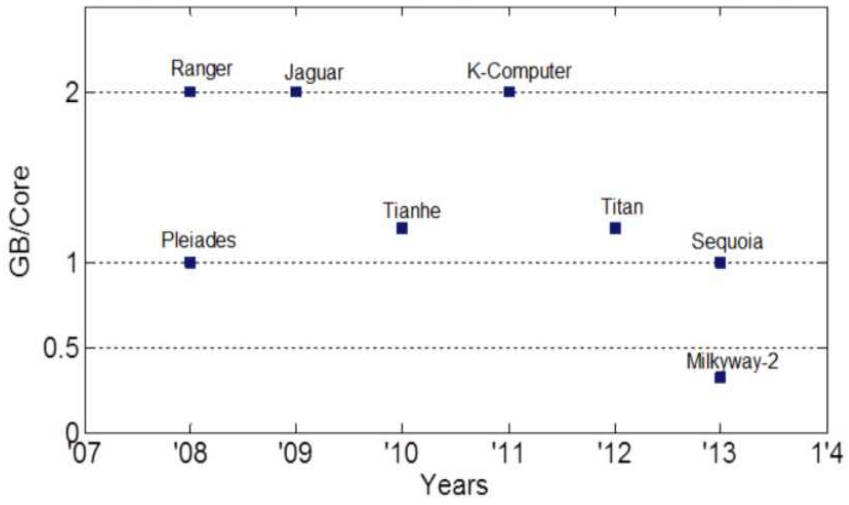
- Many/Muti-Core/Accelerator Nodes
  - Intel Xeon Phi
    - KNC 61 cores, x threads
    - KNL > 61 cores
      - Different memory models

  Power8
  - NVIDIA GPU O(1000) threads

- Key Features (Good, Bad, and Ugly)
  - Less memory per thread
  - Hierarchy of memory
    - Multiple NUMA regions
  - Nested parallelism
    - Multiple threads/core
    - Thread teams
  - UVM



Source: hpcwire.com/2016/02/10/nersc-getting-ready-knl



Source: data from top500.org

Ne

- Ma (Co
  - In
    -
    - Kn
      - –

Power8

- NVIDIA GPU

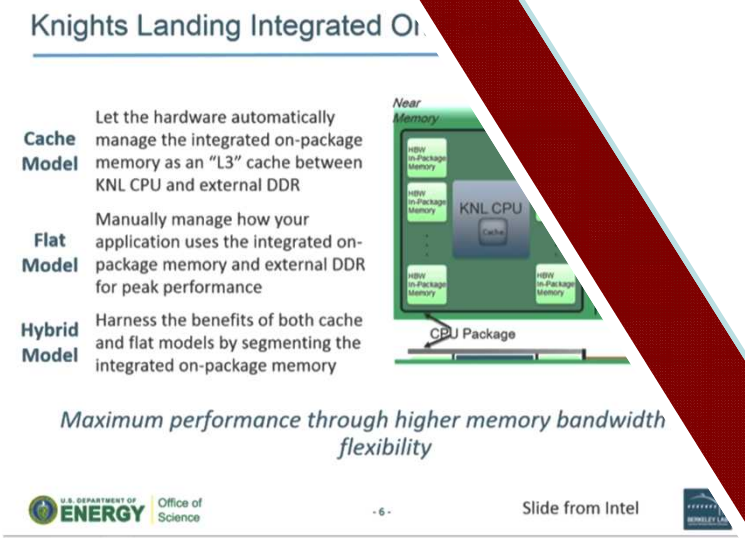
Ugly)

regions

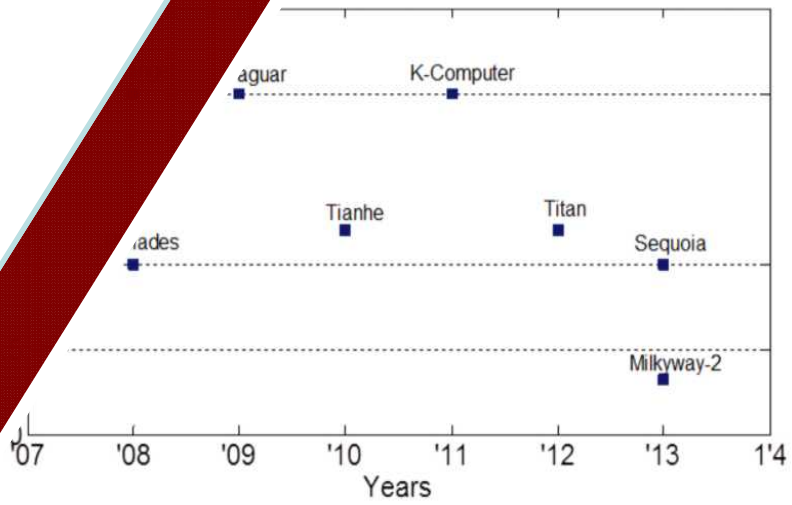
ads/core

ams

So how do we deal?
Are our current methods enough:
Data-Parallelism and
Task-Parallelism

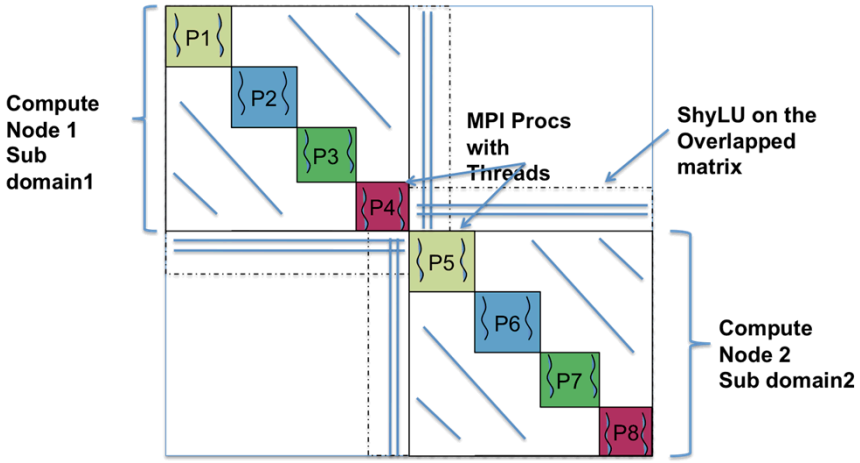


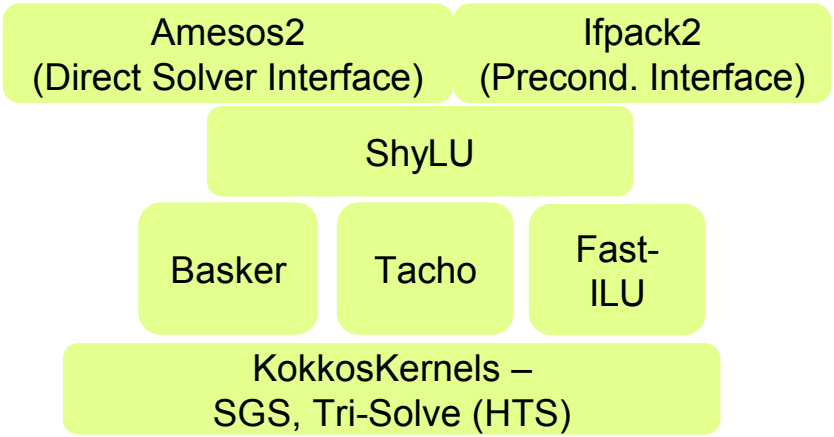
Source: hpcwire.com/2016/02/10/nersc-getting-ready-knl



Source: data from top500.org

ShyLU Effort Overview





- MPI+X based subdomain solvers
  - Decouple the notion of one MPI rank as one subdomain: Subdomains can span multiple MPI ranks each with its own subdomain solver using X or MPI+X
- **Subpackages of ShyLU (+X Part)**: Multiple Kokkos-based options for on-node parallelism
  - **Basker :** LU or ILU (t) factorization — *DATA-PARALLEL*
  - **Tacho:** Incomplete Cholesky - IC (k) — *TASK-PARALLEL*
  - **Fast-ILU:** Fast-ILU factorization for GPUs — *DATA-PARALLEL (Asynchronous)*
- **KokkosKernels (+X Part):** Coloring based Gauss-Seidel (M. Deveci), Triangular Solves
- Under active development. Jointly funded by ASC, ATDM, FASTMath, LDRD.

Data and Task Parallelism

Sandia National Laboratories

- Data-Parallelism
  - Traditional *par_for*
  - Fork-Join
  - Multiple levels with thread-teams needed for multi-threading
  - Low startup cost
  - OpenMP, Kokkos, …
- Task-Parallelism
  - Successful with irregular problems
  - Task/Futures
  - Cost for task-queue
  - Data-locality?
  - Static or Dynamic (Cost?)
  - Omps, OpenMP (4.0+), Kokkos, …

Shared Issue with Architecture Aware:

- Data layout – location
- Synchronizations / Data Sharing

Themes for Architecture Aware Solvers and Kernels : Data layouts

Sandia National Laboratories

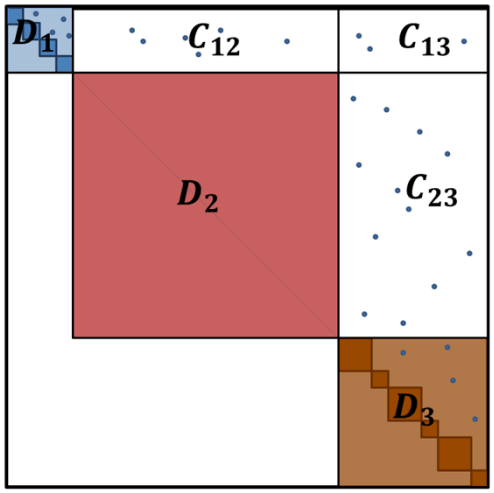
- Specialized memory layouts
  - Architecture aware data layouts
    - Coalesced memory access
    - Padding
    - Array of Structures vs Structure of Arrays
    - Kokkos based abstractions (H. C. Edwards and C. Trott)
- Two dimensional layouts for matrices
  - Allows using 2D algorithms for solvers and kernels
  - Bonus: Fewer synchronizations with 2D algorithms
  - Cons : Much more harder to design correctly
  - Better utilization of hierarchical memory like High Bandwidth Memory (HBM) in Intel Xeon Phi or NVRAM
- Hybrid layouts
  - Better for very heterogeneous problems

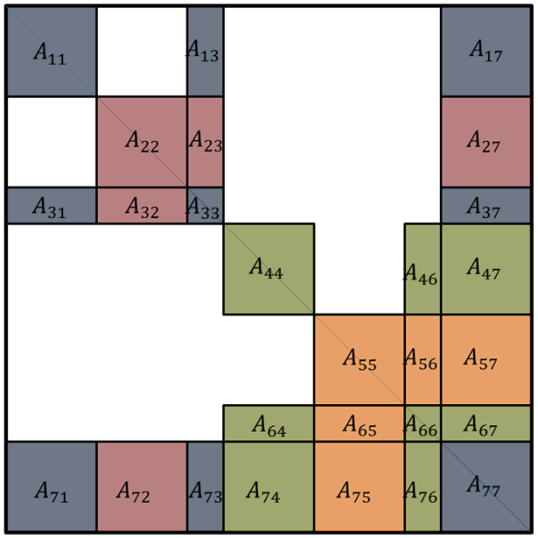
Themes for Architecture Aware Solvers and Kernels : Synchronization and Data Sharing

Sandia National Laboratories

- Synchronizations are expensive
  - 1D algorithms for factorizations and solvers, such as ND based solvers have a huge synchronization bottleneck for the final separator\
  - Impossible to do efficiently in certain architectures designed for massive data parallelism (GPUs)
  - This is true only for global synchronizations, fork/join style model.
- Fine grained synchronizations
  - Between handful of threads (teams of threads)
  - Point to Point Synchronizations instead of global synchronizations
    - Joongsoo Park et al (SC14) showed this for triangular solve
  - Thread Parallel reductions wherever possible
- Data Sharing
  - Global accesses are expensive (NUMA)
  - Atomics are cheap
    - Only when used judiciously

Basker : (I)LU factorization

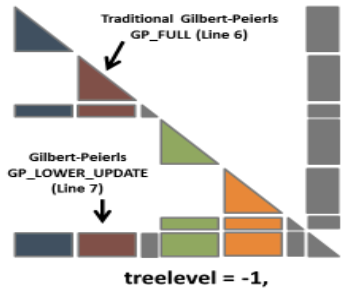




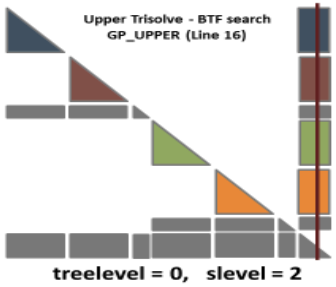
- Basker: Sparse (I)LU factorization (**w/ Joshua Booth**)
  - Block Triangular form (BTF) based LU factorization, Nested-Dissection on large BTF components
  - 2D layout of coarse and fine grained blocks
    - Previous work by Sherry Li, Rothberg, & Gupta
  - Data-Parallel, Kokkos based implementation
  - Fine-grained parallel algorithm with P2P synchronizations
  - Parallel version of Gilbert-Peirels' algorithm (or KLU)
  - Left-looking 2D algorithm requires careful synchronization between the threads
    - All reduce operations between threads to avoid atomic updates

See "*Basker: A Threaded Sparse LU Factorization Utilizing Hierarchical Parallelism and Data Layouts*" (J. Booth, S. Rajamanickam and H. Thornquist)

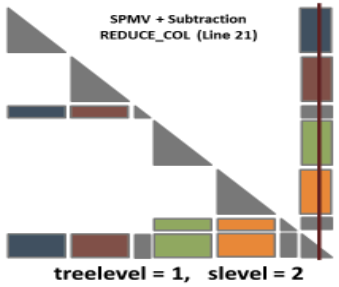
Basker : Steps in a Left looking factorization



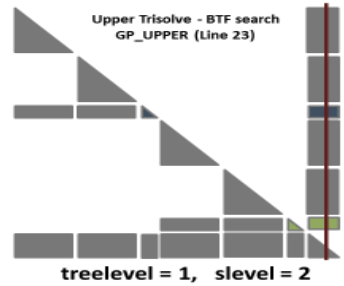
Bottom level of Dependency tree



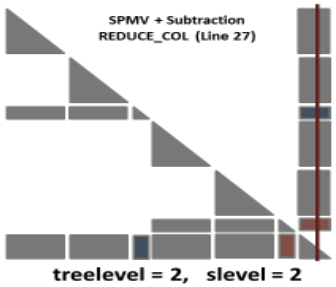
Walking from level 0, slvl is separator level



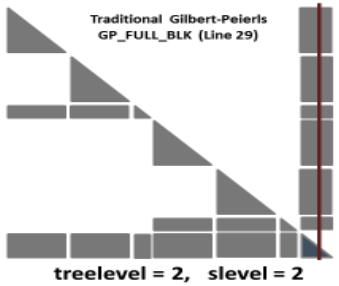
Fine grain reduction needed for level 1



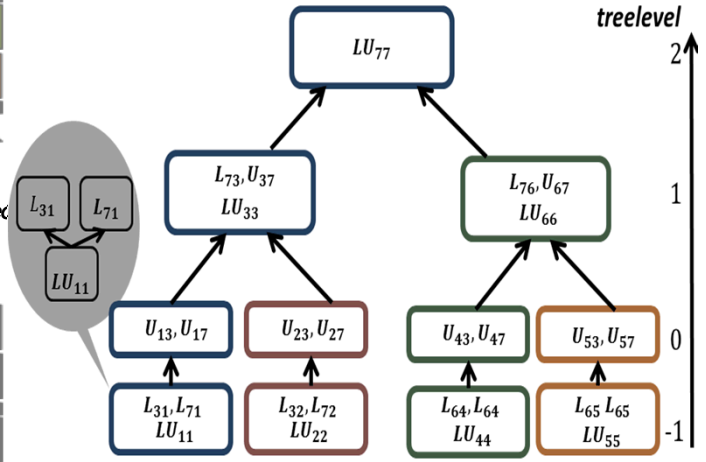
Level 1 factorization



Fine grain reduction needed for level 2

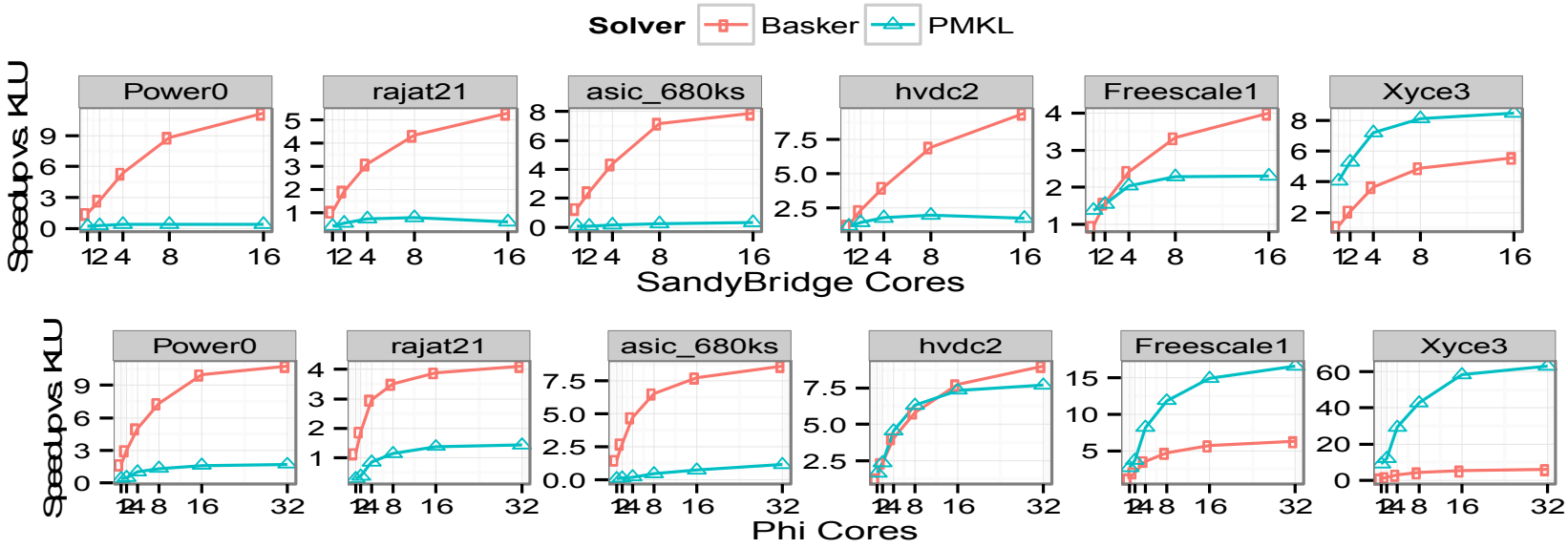


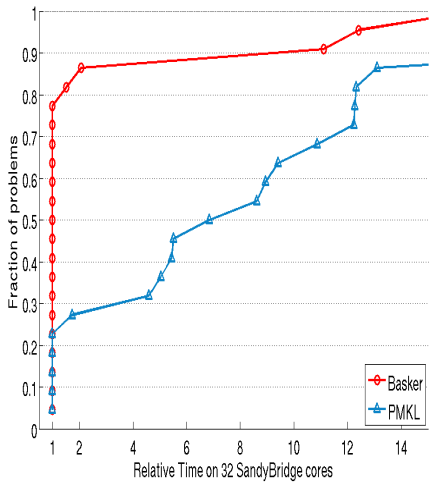
Level 2

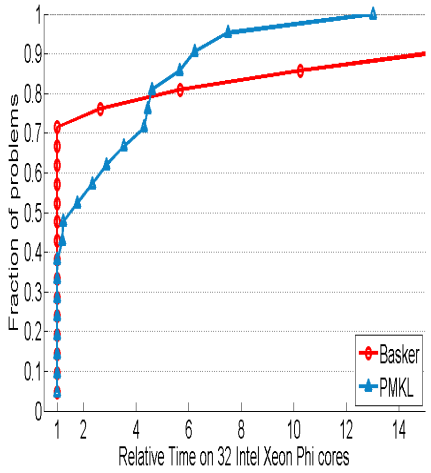


- Different Colors show different threads
- Grey means not active at any particular step
- Every left looking factorization for the final separator shown here involves four independent triangular solve, a mat-vec and updates (P2P communication), two independent triangular solves, a mat-vec and updates, and triangular solve. (Walking up the nested-dissection tree)

Basker : Performance Results

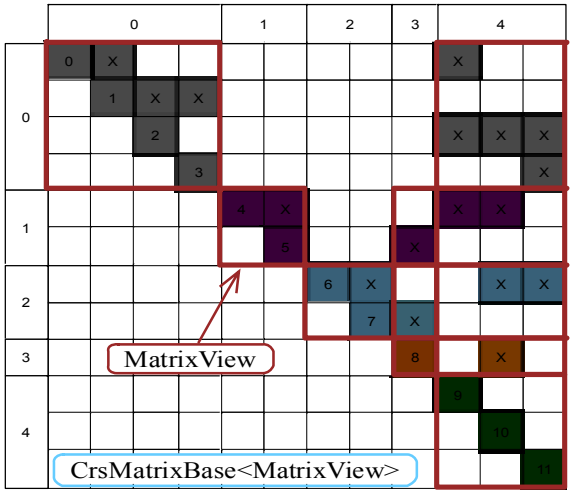






- Speedup 5.9x (CPU) and 7.4x (Xeon Phi) over KLU (Geom-Mean) and up to 53x (CPU) and 13x (Xeon Phi) over MKL
- Performance Profile for a matrix set with a high-fill and low-fill matrices shown (16 threads on CPUs /32 threads on Xeon Phi)
- Low-fill matrices Basker is consistently the better solver. High fill matrices MKL Pardiso is consistently the better solver

Tacho : Task Based Cholesky factorization



Algorithm: $A := \text{CHOL\_BLK}(A)$

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$

where A_{TL} is 0×0

while $length(A_{TL}) < length(A)$ **do**

Determine block size b

Repartition

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$$

where A_{11} is $b \times b$

$$A_{11} := \text{CHOL\_UNB}(A_{11})$$
$$A_{12} := \text{TRIU}(A_{11})^{-1} A_{12}$$
$$A_{22} := A_{22} - A_{12}^{T} A_{12}$$

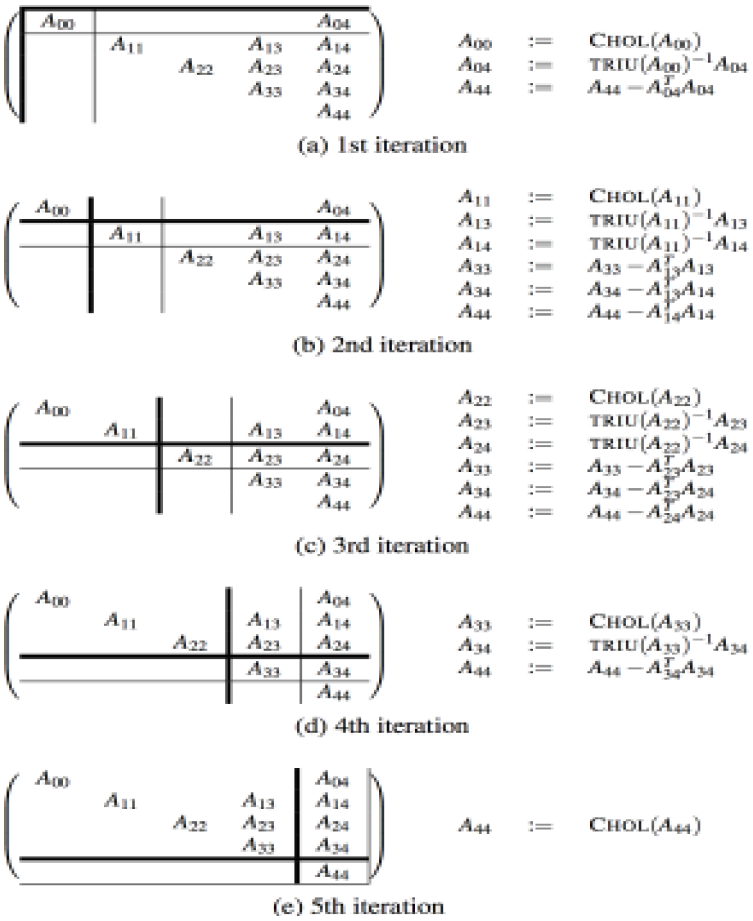
Continue with

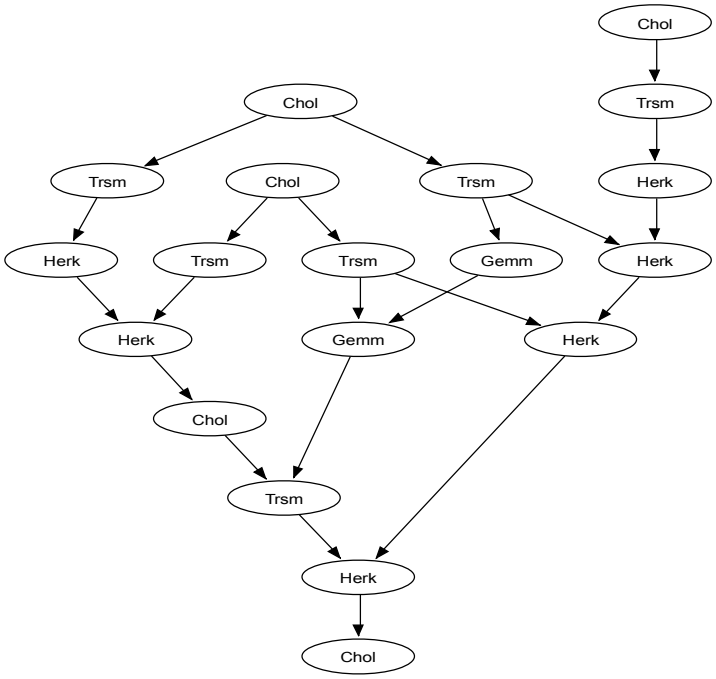
$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$$

endwhile

- Fine-grained Task-basked, Right Looking Cholesky Factorization (**w/ K. Kim**)
  - 2D layout of blocks based on nested dissection and fill pattern
  - Task-Parallel, Kokkos based implementation
  - Fine-grained parallel algorithm with synchronizations represented as a task DAG
  - Algorithm-by-blocks style algorithm
    - Originally used for parallel out-of-core factorizations (Quintana et al, Buttari et al)
    - Block based algorithm rather than scalar based algorithm
- See "*Task Parallel Incomplete Cholesky Factorization using 2D Partitioned-Block Layout*" (K. Kim, S. Rajamanickam, G. Stelle, H. C. Edwards, S. Olivier) arXiv.

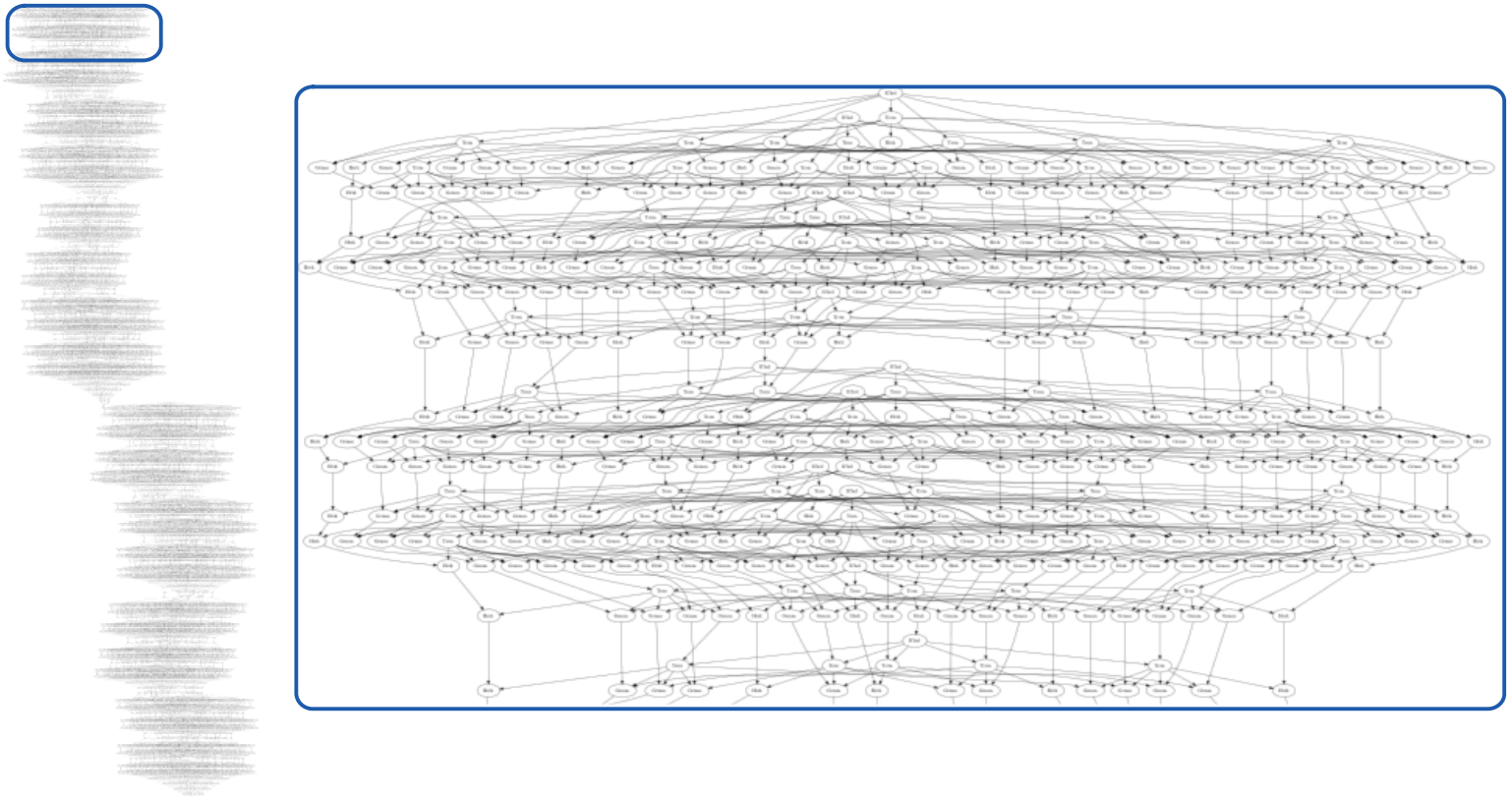
Tacho : Steps in the factorization





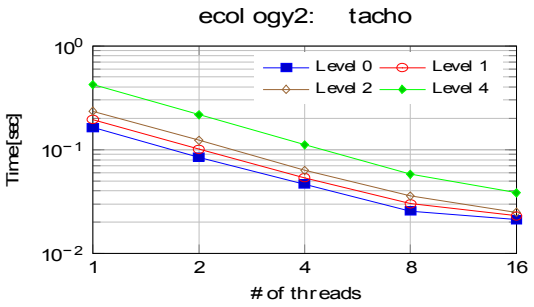
- Degree of Concurrency still depends on nested dissection ordering
- Parallelism is not tied to nested dissection ordering

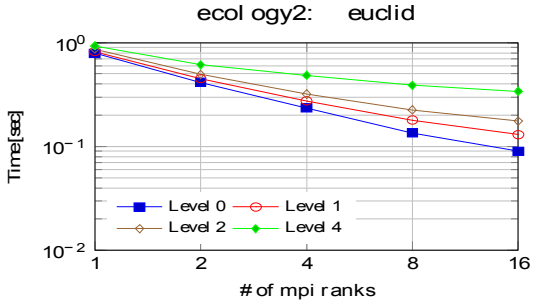
Tacho : More realistic task DAG

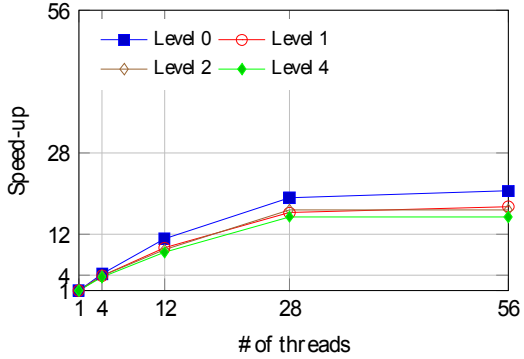


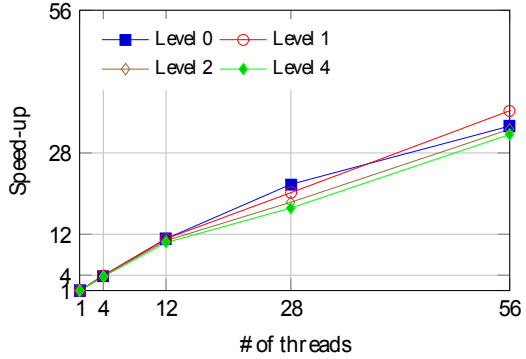
- Complete Task DAG never formed. Shown here for demonstration of the degree of concurrency.
- The concurrency is from fine-grained tasking and a 2D right looking algorithm

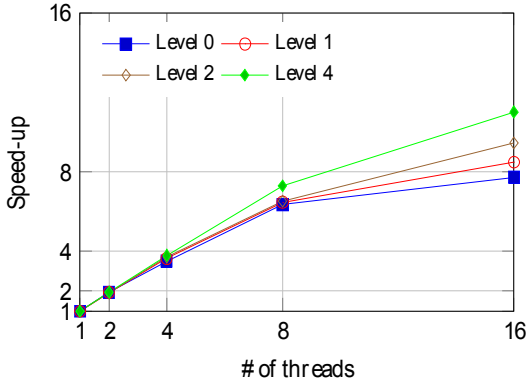
Tacho : Experimental Results

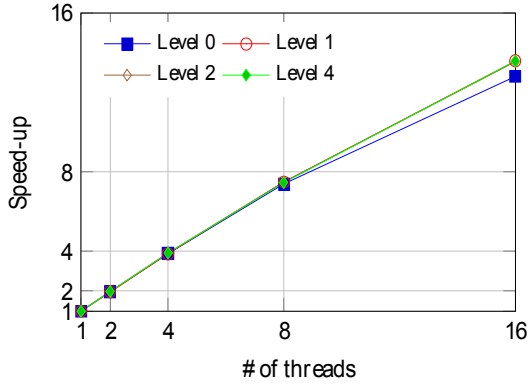










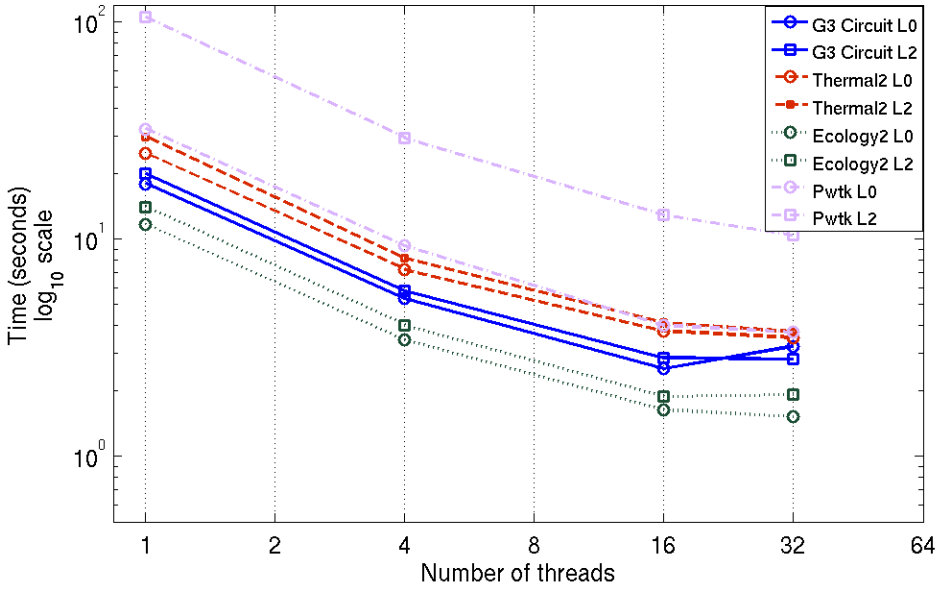


- Results shown for two matrices with different levels of fill
- Euclid results shown for reference
  - It is an MPI code, using RCM ordering (best for Euclid)
- Speedup numbers are in comparison with single threaded Cholesky
  - Small overhead for single threaded Cholesky over serial Cholesky
- Results are shown for both CPU and Xeon Phi architectures
- The two matrices are chosen for very different nnz/n.

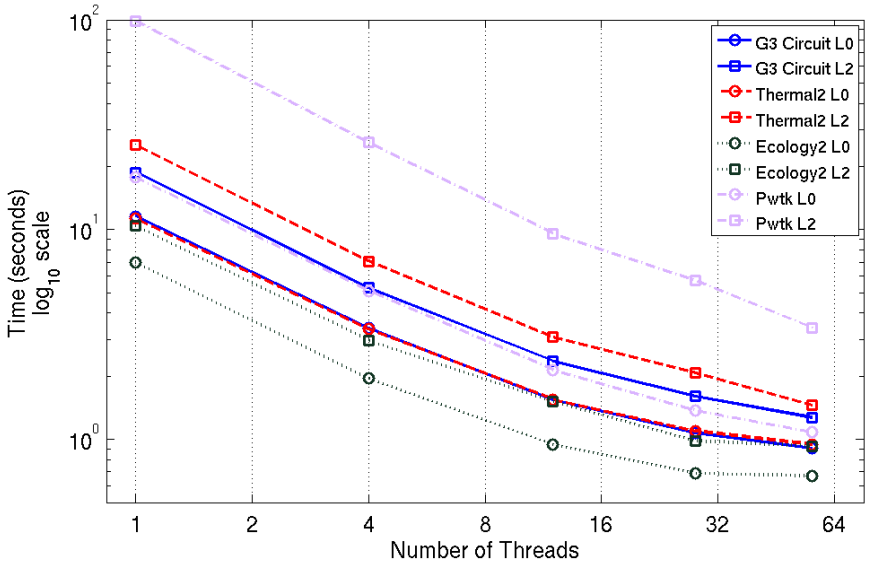
Comparison of ILUK and ICK, Data/Tasking Parallelism

Sandia National Laboratories

Time, Basker (ILUK) – Intel Xeon Phi



Time, Tacho (ICK) – Intel Xeon Phi

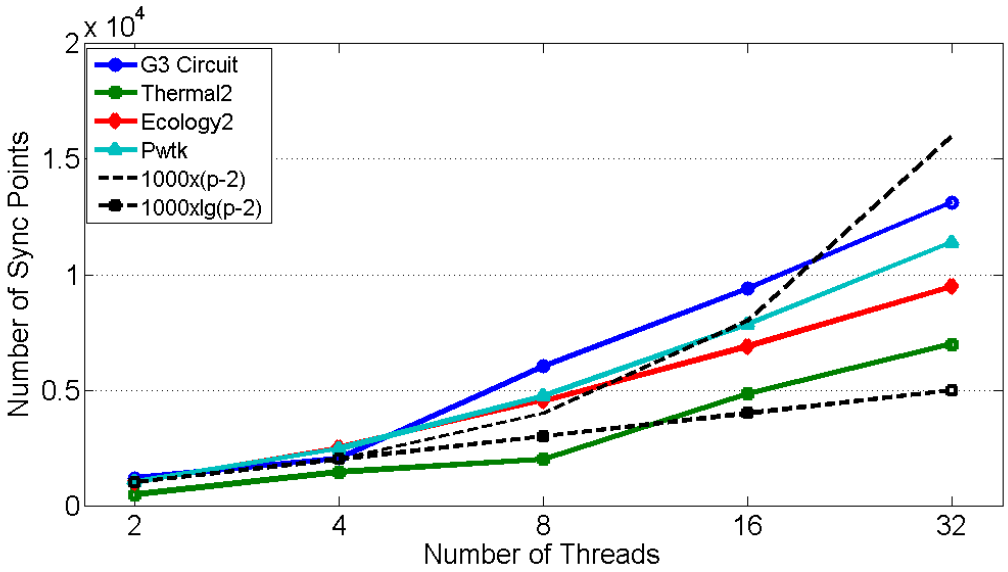


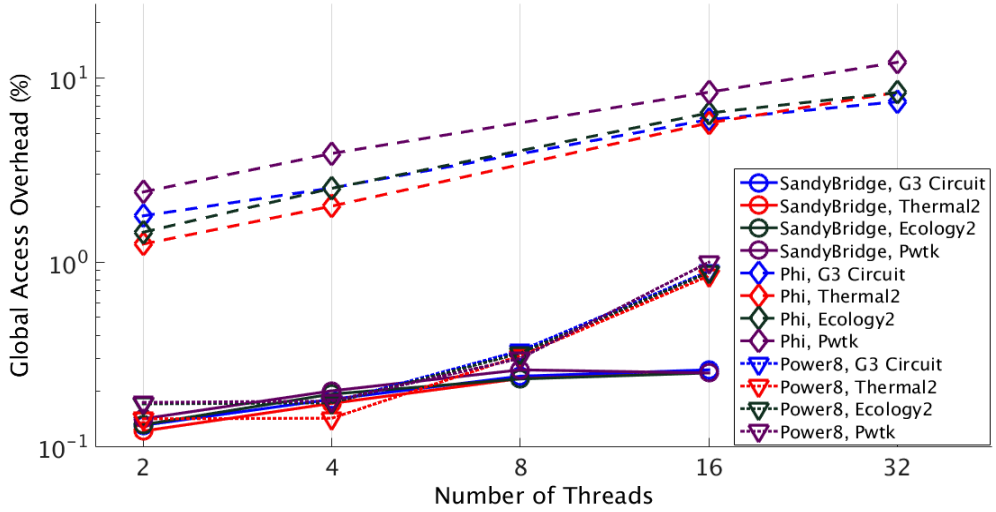
- Detail account of this analysis to be presented at HIPS workshop IPDPS 2016 (Booth, Kim, and Rajamanickam).
- Time_Tacho = Symbolic Time + Numeric Time Time_Basker = 2xNumeric Time
- On x86-Intel Sandy Bridge, similar observed performance
- On Intel Xeon Phi, Basker (ILUK) with data-parallelism and global accesses to find sparsity pattern suffers

Comparison of ILUK and ICK, Data/Tasking Parallelism (Impact of Synchronization / Data Sharing)

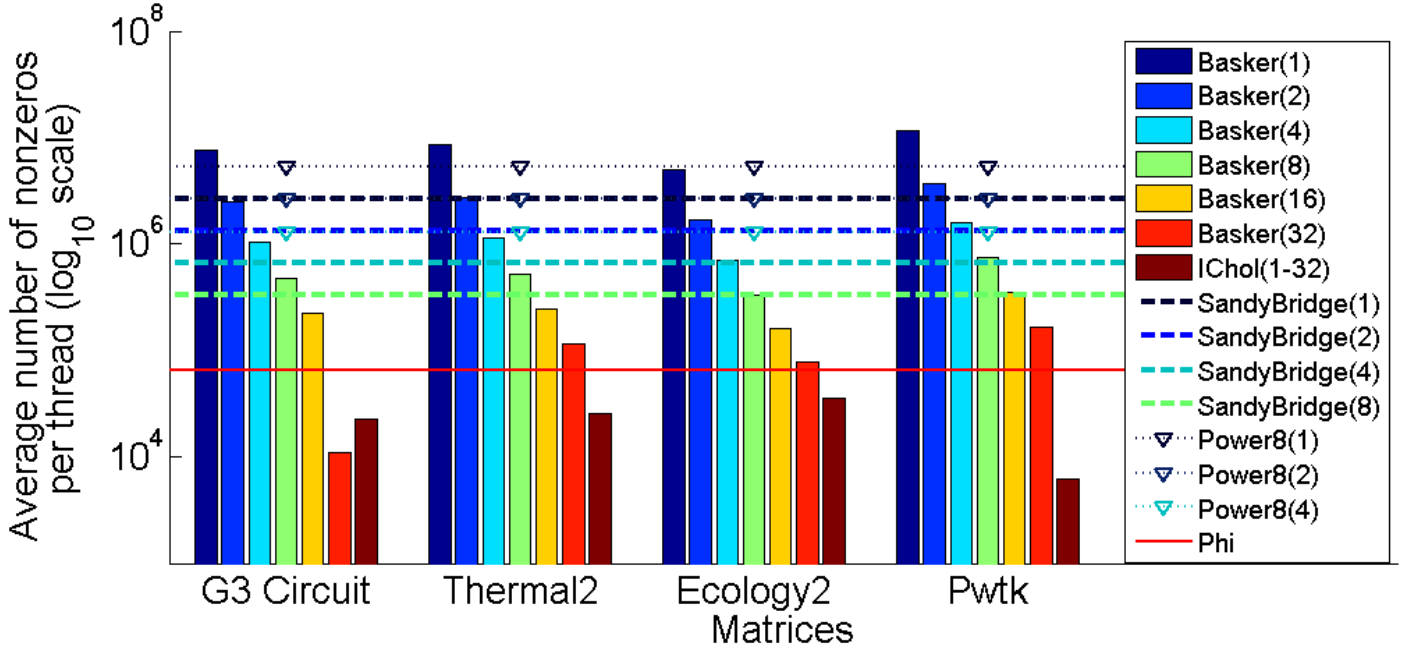
Sandia National Laboratories

- As number of threads increase, so will the number of synchronization in data-parallelism case
- Global Access Overhead = Time Global Access/ Total Time x 100
- Any data sharing via a synchronization or atomics cost will grow
- The tasking overhead:
  - Task generation
  - Scheduling
  - Context switching
- Synch and sharing time maybe > tasking overhead for large thread counts





Comparison of ILUK and ICK, Data/Tasking Parallelism (Impact of Data Layout)



- Average number of nonzero each thread is working on
- Would like the “chunk of data” being worked on to fit into cache
- Trade-off, number of tasks generated (TP) and number of synchronizations (DP)
- Intel Xeon Phi / Power8 have segmented shared cache
- Task Parallelism breaking tasks into a uniform size fits the cache structure
- Data Parallelism breaking into pieces to fit number of threads and limit synchronizations may fall out of shared cache structure on many-core systems

Asynchronous ILU factorizations

$$(LU)_{ij} = a_{ij}, \quad (i,j) \in S$$

- The factorization is exact on the sparsity pattern (S)
  - ILU methods were interpreted this way before the current "traditional" ILU (Varga 1959, Oliphant 1961, Buleev 1960)
- Chow & Patel (2015) uses this property to compute ILU factorizations

Solve $\begin{matrix} l_{ij}, & i > j, & (i,j) \in S \\ u_{ij}, & i \leq j, & (i,j) \in S. \end{matrix}$ with the constraints $\sum_{k=1}^{\min(i,j)} l_{ik} u_{kj} = a_{ij}, \quad (i,j) \in S.$

- The equations are non-linear with more equations than the original !
- Equations can be solved in a fine-grained inexact manner with a good initial guess
- *Same idea can be applied to tri-solve*
- Implemented for ShyLU/FAST-ILU by Patel, Rajamanickam, and Boman

Asynchronous ILU factorization + Tri Solves vs Exact ILU factorization

Sandia National Laboratories

FastILU
10 sweeps,
RCM ordering
GPUs

	0	1	2	3	4	5
thermal2	1312	869	692	635	630	626
af_shell3	*	*	*	*	*	*
ecology2	1708	1082	832	761	738	723
apache2	967	541	326	299	293	294
offshore	334	*	*	*	*	*
G3_circuit	860	524	426	360	312	254
Parabolic_fem	334	271	255	249	263	292

Exact ILU

	0	1	2	3	4	5
thermal2	1934	1225	856	637	507	440
af_shell3	1248	788	583	462	369	309
ecology2	1625	988	696	576	467	414
apache2	1294	619	394	289	235	188
offshore	485	*	*	*	*	*
G3_circuit	1414	757	546	421	341	303
Parabolic_fem	313	238	164	129	106	91

Stabilization Techniques

- FastILU Factorization
  - A lot of concurrency in the GPUs
  - Almost in the the nonlinear Jacobi regime than in the nonlinear Gauss Seidel regime
  - Triangular Solves are even bigger problem
- Common technique to stabilize convergence : damping or underelaxation

$$x^{(k+1)} = (1-\omega)x^{(k)} + \omega G(x^{(k)}).$$

- Damping factor 0-1 allows controlling the effect of numerical overflow and enormous concurrency
- Choosing the damping factor is much more trickier

Restart/Blocking Techniques

Sandia National Laboratories

- Use a warm restart for ILU(k) k >0
  - Instead of using A as the initial guess use few sweeps of ILU(k-1) as the initial guess
  - Continuation like method, recursively find initial guess with lower fill levels
- Block Jacobi iteration instead of Jacobi iteration on the triangular solves with small block sizes for each thread
  - Need to be careful about performance vs convergence
  - Assigning blocks to a warp / thread block balances both
  - Kokkos allows teams of threads to do this naturally
- Another stabilization approach: Manteuffel shifting didn't help these methods

Asynchronous ILU factorization + Tri Solves vs Exact ILU factorization

Sandia National Laboratories

FastILU
10 sweeps,
RCM ordering
GPUs, damping
Factor = 0.5

	0	1	2	3	4	5
thermal2	1343	924	840	815	819	811
af_shell3	901	653	565	589	554	599
ecology2	1704	1103	925	910	893	922
apache2	1043	629	432	484	427	497
offshore	350	211	184	175	172	172
G3_circuit	904	607	512	471	431	410
Parabolic_fem	356	328	295	288	285	286

Exact ILU

	0	1	2	3	4	5
thermal2	1934	1225	856	637	507	440
af_shell3	1248	788	583	462	369	309
ecology2	1625	988	696	576	467	414
apache2	1294	619	394	289	235	188
offshore	485	*	*	*	*	*
G3_circuit	1414	757	546	421	341	303
Parabolic_fem	313	238	164	129	106	91

Asynchronous ILU factorization + Tri Solves vs Exact ILU factorization

Sandia National Laboratories

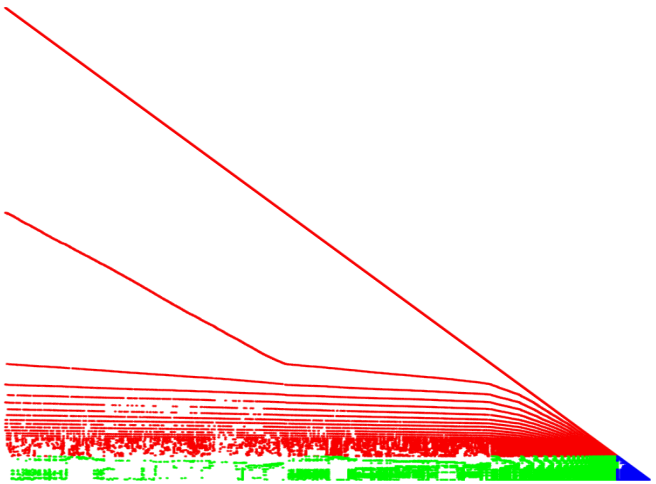
FastILU
10 sweeps,
RCM ordering
GPUs, damping
Factor = 0.5,
Continuation
guess

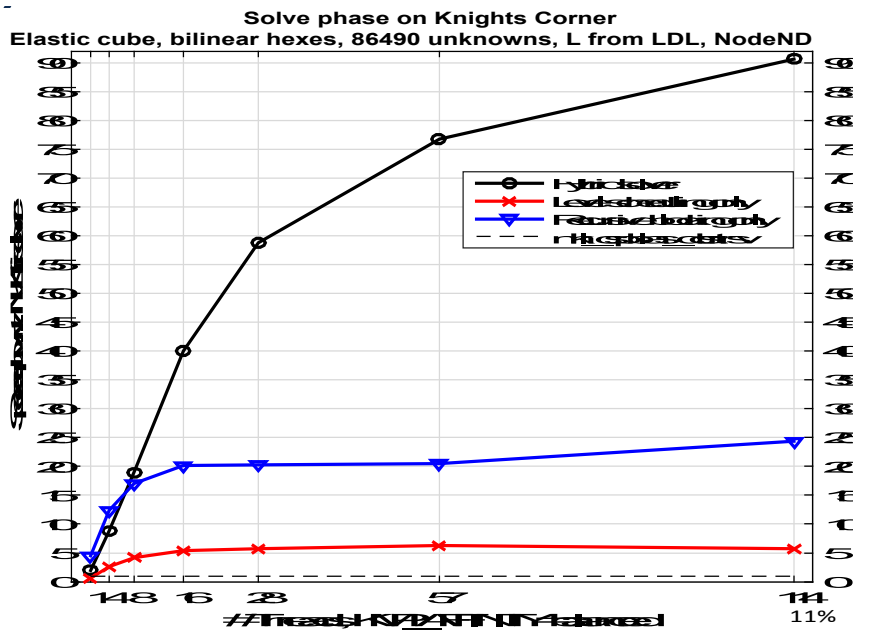
	0	1	2	3	4	5
thermal2	1343	904	727	646	623	608
af_shell3	902	666	569	534	547	424
ecology2	1704	1114	879	799	759	725
apache2	1043	592	370	291	267	267
offshore	350	205	178	178	282	274
G3_circuit	903	567	498	419	357	291
Parabolic_fem	358	296	253	246	251	256

Exact ILU

	0	1	2	3	4	5
thermal2	1934	1225	856	637	507	440
af_shell3	1248	788	583	462	369	309
ecology2	1625	988	696	576	467	414
apache2	1294	619	394	289	235	188
offshore	485	*	*	*	*	*
G3_circuit	1414	757	546	421	341	303
Parabolic_fem	313	238	164	129	106	91

HTS Triangular Solve





- Hybrid Triangular Solves (**A. Bradley**)
  - Algorithm is a combination of level-set triangular solve and recursive blocked triangular solve
  - Implementation based on P2P synchronizations similar to Park et al.
  - Level-Set portion of the algorithm works on the low-fill portions of the problem
  - Recursive blocked algorithm is used in the denser portions of the algorithm (separators, supernodes)
  - Scales well on CPU and Xeon Phi architectures

Summary

Sandia National Laboratories

- ShyLU Efforts
  - Trying many different directions in order to provide fast solver (Mostly successful with a lot of algorithm and implementation tweaking)
  - A collection of different methods and different backend all coded in a similar language and library in order to aid fair comparison
  - A real MPI+X future option
- Data Parallelism
  - Can be useful if unique irregular structure exist that can be exploited (Basker LU with BTF)
  - Can have a much lower overhead than tasking for few threads and little work (low fill-in Basker ILUK)
  - Need light-weight synchronization P2P (Basker and HTS)
- Tasking Parallelism
  - Flexible
  - Can break task into cache friendly sizes
  - Any tasking overhead can be amortized with increasing thread counts, ideal since we have so many threads
- Asynchronous
  - A real viable option on GPU
  - More work in this section on this to come!

Sandia National Laboratories

Questions ?