Exceptional service in the national interest

Sandia National Laboratories

Task-parallel Sparse Incomplete Cholesky Factorization using Kokkos Portable APIs.

Kyungjoo Kim   Sivasankaran Rajamanickam
H. Carter Edwards   Stephen L. Olivier   George Stelle

Collaborators : E. Boman   J. Booth   A. Bradley   C. Dohrmann   S. Hammond

Center for Computing Research, Sandia National Labs

Trilinos User Group Meeting, October 26-29, 2015

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. SAND NO. SAND2015-9269 C

Overview

Kokkos Portable Task Parallel Programming Model

Task-parallel (In)complete Cholesky Factorization

Numerical Examples

Conclusion

LDRD: Unified Task-Data Manycore Parallelism Kokkos and Qthreads

Sandia National Laboratories

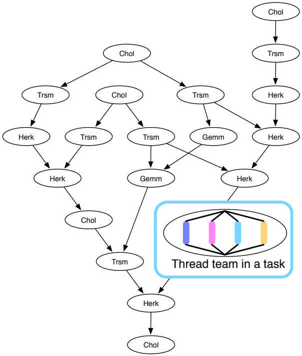
Deep hierarchical features of current hardware

- Multi-socket, multi-processor, multi(or many)-core, multiple hardware threads.
- Multiple NUMA regions, multiple levels of caches, segmented and shared cache.

Need to expose more fine-grained parallelism

- Task parallelism is suitable for irregular problems:
  e.g., producer/consumer, recursive algorithms.
- Kokkos addresses high-level abstractions for data parallelism.

→ Nested data-parallelism within a task provides better locality exploiting hardware threads.



Research highlights

- Abstractions harnessing multiple tasking backends to heterogeneous devices.
- Dependence driven asynchronous task execution with data-parallel thread team.
- Wait-free respawn task mechanism,
  e.g., a task on GPUs cannot wait on dependence.
- Mini-apps (*e.g.*, sparse factorization) to support and evaluate development.

Kokkos Portable Task API

- TaskPolicy coordinates how and where tasks are executed
  e.g., `create`, `add_dependence`, `spawn` (or `respawn`), `wait`;
- Future is a handle for tasks and allows dependence among them.

```
void SimpleTask() {
  typedef Kokkos::Threads exec_space; //Serial, Threads, Qthread

  Kokkos::TaskPolicy<exec_space> policy;
  Kokkos::Future<int> f = policy.create( Functor<exec_space>() );

  policy.spawn( f );

  Kokkos::wait( f );
}
```

- Functor includes a user-defined function body and associated data sets.

```
class Functor<exec_space> {
public:
  Kokkos::View<exec_space> data;

  void apply( int &r_val ) {
    r_val = doSomething( data );
  }
};
```

- DAG of tasks is implicitly formed to guide asynchronous task execution.

```
void SimpleDAG() {
  typedef Kokkos::Threads exec_space;

  Kokkos::TaskPolicy<exec_space> policy;
  Kokkos::View<exec_space> x, y;  // data sets for task

  auto /* future */ fx = policy.create( Functor<exec_space>( x ) );
  auto /* future */ fy = policy.create( Functor<exec_space>( y ) );

  // dependence of tasks is expressed before spawning
  policy.add_dependence( fx, fy ); // fx is scheduled after fy

  policy.spawn( fx ); // wait for now
  policy.spawn( fy ); // may immediately execute

  Kokkos::wait( policy ); // wait for all tasks to complete
}
```

Nested data parallelism with a team of threads

```
class Functor<exec_space> {
public:
  Kokkos::View<exec_space> data;

  // member is mpi-like thread communicator interface
  // i.e., member.{team_rank,team_size,team_barrier,team_reduce}
  void apply( const policy_type::member_type &member, int &r_val ) {
    Kokkos::parallel_for( TeamThreadRange( member, data.size() ),
                          [&](const int i ) {
        // different indexing may be required for different
        // execution space e.g., GPU interleaved data layout
        int id = Index<exec_space>( member, i );
        doSomething( data(id) );
      } );
  }
};

void SimpleTaskData() {
  typedef Kokkos::Threads exec_space;

  Kokkos::TaskPolicy<exec_space> policy;
  auto /* future */ f = policy.create_team( Functor<exec_space>() );

  policy.spawn( f );

  Kokkos::wait( f );
}
```

Task-parallel (In)complete Cholesky Factorization

Standard procedure

1. Fill-reduced (or band-reduced) ordering: Scotch.
2. Symbolic factorization: Hysom and Pothen[1].
3. **Numeric factorization.**

Design considerations for task-parallelism

- Structure-based (in)complete factorization; fills are statically determined.
- A set of self-contained data within a task.
- Cache-friendly numeric kernels.
- Separation of concerns (concurrency is separated from parallelism):
  - Algorithm decomposes factorization into subproblems and provides dependence among them.
  - Runtime schedules tasks to parallel compute units.

→ **Objective: portable performance on most of heterogeneous architectures.**

1 D.Hysom and A.Pothen, Level-based incomplete LU factorization: Graph model and algorithms, 2002.

Algorithms-by-blocks

Sandia National Laboratories

- Originally developed for distributed parallel *out-of-core* matrix computations.
- Converts basic computing units from *scalar* to *blocks*.
- Used for thread-level task parallelism[2,3]: **asynchronous tasking** and **efficient level 3 BLAS**.

Algorithm: $A := \textrm{CHOL\_UNB}(A)$

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**

Repartition

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

where α_{11} is a scalar

$$\begin{aligned} \alpha_{11} &:= \sqrt{\alpha_{11}} \\ a_{12}^T &:= a_{12}^T / \alpha_{11} \\ A_{22} &:= A_{22} - a_{12} a_{12}^T \end{aligned}$$

Continue with

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

endwhile

2 G.Quintana-Ortí *et al.*, Programming Matrix Algorithms-by-blocks for Thread-level Parallelism, 2009.

3 A.Buttari *et al.*, A Class of Parallel Tiled Linear Algebra Algorithms for Multicore Architectures, 2009.

Algorithms-by-blocks

- Originally developed for distributed parallel *out-of-core* matrix computations.
- Converts basic computing units from *scalar* to *blocks*.
- Used for thread-level task parallelism[2,3]: **asynchronous tasking** and **efficient level 3 BLAS**.

Algorithm: $A := \textsc{Chol\_blk}(A)$

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 $b \times b$

$$\begin{aligned} A_{11} &:= \textsc{Chol\_unb}(A_{11}) \\ A_{12} &:= \textsc{triu}(A_{11})^{-1} A_{12} \\ A_{22} &:= A_{22} - A_{12}^{T} A_{12} \end{aligned}$$

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

k-iteration

$$\left(\begin{array}{cc|c|cc} \bar{A}_{00} & \cdots & \bar{A}_{0K} & \cdots & \bar{A}_{0,N-1} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ \hline \bar{A}_{K0} & \cdots & \bar{A}_{KK} & \cdots & \bar{A}_{K,N-1} \\ \hline \vdots & \ddots & \vdots & \ddots & \vdots \\ \bar{A}_{N-1,0} & \cdots & \bar{A}_{N-1,K} & \cdots & \bar{A}_{N-1,N-1} \end{array}\right)$$

$$A_{11} := \textsc{Chol}(\bar{A}_{kk})$$

$$\begin{aligned} A_{12} &:= \bar{A}_{kk}^{-1} \left(\bar{A}_{k,k+1} \cdots \bar{A}_{k,n-1}\right) \\ &= \left(\bar{A}_{kk}^{-1}\bar{A}_{k,k+1} \cdots \bar{A}_{kk}^{-1}\bar{A}_{k,n-1}\right) \end{aligned}$$

$$\begin{aligned} A_{22} &:= \begin{pmatrix} \bar{A}_{k+1,k+1} & \cdots & \bar{A}_{k+1,n-1} \\ & \ddots & \vdots \\ sym & & \bar{A}_{n-1,n-1} \end{pmatrix} - \begin{pmatrix} \bar{A}_{k+1,k} \\ \vdots \\ \bar{A}_{n-1,k} \end{pmatrix} \begin{pmatrix} \bar{A}_{k,k+1} & \cdots & \bar{A}_{k,n-1} \end{pmatrix} \\ &= \begin{pmatrix} \bar{A}_{k+1,k+1} - \bar{A}_{k+1,k}\bar{A}_{k,k+1} & \cdots & \bar{A}_{k+1,n-1} - \bar{A}_{k+1,k}\bar{A}_{k,n-1} \\ & \ddots & \vdots \\ sym & & \bar{A}_{n-1,n-1} - \bar{A}_{n-1,k}\bar{A}_{k,n-1} \end{pmatrix} \end{aligned}$$

2 G.Quintana-Ortí *et al.*, Programming Matrix Algorithms-by-blocks for Thread-level Parallelism, 2009.

3 A.Buttari *et al.*, A Class of Parallel Tiled Linear Algebra Algorithms for Multicore Architectures, 2009.

Algorithms-by-blocks

- Originally developed for distributed parallel *out-of-core* matrix computations.
- Converts basic computing units from *scalar* to *blocks*.
- Used for thread-level task parallelism[2,3]: **asynchronous tasking** and **efficient level 3 BLAS**.

Algorithm: $A := \textsc{Chol\_blk}(A)$

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 $b \times b$

$$\begin{aligned} A_{11} &:= \textsc{Chol\_unb}(A_{11}) \\ A_{12} &:= \textsc{triu}(A_{11})^{-1} A_{12} \\ A_{22} &:= A_{22} - A_{12}^T A_{12} \end{aligned}$$

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

k-iteration

$$\left(\begin{array}{cc|c|cc} \bar{A}_{00} & \cdots & \bar{A}_{0K} & \cdots & \bar{A}_{0,N-1} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ \hline \bar{A}_{K0} & \cdots & \bar{A}_{KK} & \cdots & \bar{A}_{K,N-1} \\ \hline \vdots & \ddots & \vdots & \ddots & \vdots \\ \bar{A}_{N-1,0} & \cdots & \bar{A}_{N-1,K} & \cdots & \bar{A}_{N-1,N-1} \end{array}\right)$$

$$A_{11} := \textsc{Chol}(\bar{A}_{kk})$$

Tri-Solve

$$\begin{aligned} A_{12} &:= \bar{A}_{kk}^{-1}\left(\bar{A}_{k,k+1} \cdots \bar{A}_{k,n-1}\right) \\ &= \left(\bar{A}_{kk}^{-1}\bar{A}_{k,k+1} \cdots \bar{A}_{kk}^{-1}\bar{A}_{k,n-1}\right) \end{aligned}$$

$$\begin{aligned} A_{22} &:= \begin{pmatrix} \bar{A}_{k+1,k+1} & \cdots & \bar{A}_{k+1,n-1} \\ & \ddots & \vdots \\ sym & & \bar{A}_{n-1,n-1} \end{pmatrix} - \begin{pmatrix} \bar{A}_{k+1,k} \\ \vdots \\ \bar{A}_{n-1,k} \end{pmatrix} \begin{pmatrix} \bar{A}_{k,k+1} & \cdots & \bar{A}_{k,n-1} \end{pmatrix} \\ &= \begin{pmatrix} \bar{A}_{k+1,k+1} - \bar{A}_{k+1,k}\bar{A}_{k,k+1} & \cdots & \bar{A}_{k+1,n-1} - \bar{A}_{k+1,k}\bar{A}_{k,n-1} \\ & \ddots & \vdots \\ sym & & \bar{A}_{n-1,n-1} - \bar{A}_{n-1,k}\bar{A}_{k,n-1} \end{pmatrix} \end{aligned}$$

General Matrix-Matrix mult.

Hermitian Rank K-update

2 G.Quintana-Ortí *et al.*, Programming Matrix Algorithms-by-blocks for Thread-level Parallelism, 2009.

3 A.Buttari *et al.*, A Class of Parallel Tiled Linear Algebra Algorithms for Multicore Architectures, 2009.

Sparse Cholesky-by-blocks

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

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**

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 1×1

genTaskChol(A_{11})
genTaskTrsm(A_{11}, A_{12})
genTaskHerk(A_{12}, A_{22})

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

```
function genTaskChol :
  Future f = create( Chol, A11(0,0) )
  add_dependence( f, A11(0,0).getFuture() )
  A11(0,0).setFuture( f )
  spawn( f )

function genTaskTrsm :
  for j in A12.nnz()
    Future f = create( Trsm, A11(0,0), A12(0,j))
    add_dependence( f, A11(0,0).getFuture() )
    add_dependence( f, A12(0,j).getFuture() )
    A12(0,j).setFuture( f )
    spawn( f )

function genTaskHerk :
  for i in A12.nnz()
    for j in A12.nnz()
      if exist( A22(i,j) )
        Future f = create( i==j ? Herk : Gemm,
                           A12(0,i), A12(0,j),
                           A22(i,j) )
        add_dependence( f, A12(0,i).getFuture() )
        add_dependence( f, A12(0,j).getFuture() )
        add_dependence( f, A22(i,j).getFuture() )
        A21(i,j).setFuture( f )
        spawn( f )
```

- Degree of concurrency still depends on nested dissection ordering.
- Parallelism is not strictly tied with the nested dissection tree.
- Each block records `future` and dependence is explicitly described from the algorithm.

2D Partitioned-block Matrix

	0	1	2	3	4	5	6	7	8	9	10	11
0	0	X			X							
1		1	X			X						
2			2	X			X					
3				3				X				
4					4	X			X			
5						5	X			X		
6							6	X			X	
7								7				X
8									8	X		
9										9	X	
10											10	X
11	CrsMatrixBase<Scalar>											11

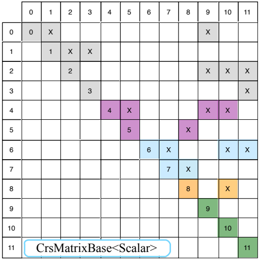
Natural ordering from a 2D mesh

Hierarchical (recursive) definition of matrices

- `CrsMatrixBase` contains sparse data arrays *i.e.*, row pointers, column indices, value array.
- `MatrixView` defines a rectangular region *i.e.*, view offsets and dimensions;
    - light-weight object with meta data only.

```
typedef CrsMatrixBase<value_type := scalar> ScalarMatrix;
typedef MatrixView<base_object := ScalarMatrix> ViewOnScalarMatrix;

typedef CrsMatrixBase<value_type := ViewOnScalarMatrix> BlockMatrix;
typedef MatrixView<base_object := BlockMatrix> ViewOnBlockMatrix;
```

2D Partitioned-block Matrix



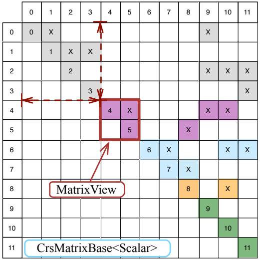
Nested dissection ordering by Scotch

Hierarchical (recursive) definition of matrices

- CrsMatrixBase contains sparse data arrays
  i.e., row pointers, column indices, value array.
- MatrixView defines a rectangular region
  i.e., view offsets and dimensions;
  - light-weight object with meta data only.

```
typedef CrsMatrixBase<value_type := scalar> ScalarMatrix;
typedef MatrixView<base_object := ScalarMatrix> ViewOnScalarMatrix;

typedef CrsMatrixBase<value_type := ViewOnScalarMatrix> BlockMatrix;
typedef MatrixView<base_object := BlockMatrix> ViewOnBlockMatrix;
```

2D Partitioned-block Matrix



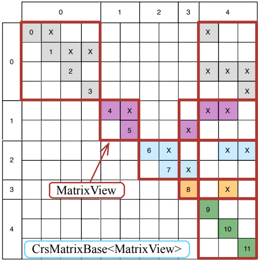
Matrix view

Hierarchical (recursive) definition of matrices

- `CrsMatrixBase` contains sparse data arrays *i.e.*, row pointers, column indices, value array.
- `MatrixView` defines a rectangular region *i.e.*, view offsets and dimensions;
  - light-weight object with meta data only.

```
typedef CrsMatrixBase<value_type := scalar> ScalarMatrix;
typedef MatrixView<base_object := ScalarMatrix> ViewOnScalarMatrix;

typedef CrsMatrixBase<value_type := ViewOnScalarMatrix> BlockMatrix;
typedef MatrixView<base_object := BlockMatrix> ViewOnBlockMatrix;
```

2D Partitioned-block Matrix



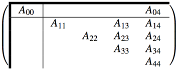
Matrix of blocks

Hierarchical (recursive) definition of matrices

- `CrsMatrixBase` contains sparse data arrays *i.e.*, row pointers, column indices, value array.
- `MatrixView` defines a rectangular region *i.e.*, view offsets and dimensions;
  - light-weight object with meta data only.

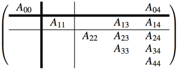
```
typedef CrsMatrixBase<value_type := scalar> ScalarMatrix;
typedef MatrixView<base_object := ScalarMatrix> ViewOnScalarMatrix;

typedef CrsMatrixBase<value_type := ViewOnScalarMatrix> BlockMatrix;
typedef MatrixView<base_object := BlockMatrix> ViewOnBlockMatrix;
```

Example



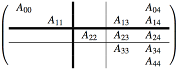
$$
\begin{aligned}
A_{00} &:= \text{CHOL}(A_{00}) \\
A_{04} &:= \text{TRIU}(A_{00})^{-1}A_{04} \\
A_{44} &:= A_{44} - A_{04}^T A_{04}
\end{aligned}
$$

(a) 1st iteration



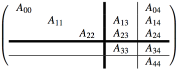
$$
\begin{aligned}
A_{11} &:= \text{CHOL}(A_{11}) \\
A_{13} &:= \text{TRIU}(A_{11})^{-1}A_{13} \\
A_{14} &:= \text{TRIU}(A_{11})^{-1}A_{14} \\
A_{33} &:= A_{33} - A_{13}^T A_{13} \\
A_{34} &:= A_{34} - A_{13}^T A_{14} \\
A_{44} &:= A_{44} - A_{14}^T A_{14}
\end{aligned}
$$

(b) 2nd iteration



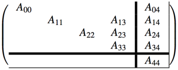
$$
\begin{aligned}
A_{22} &:= \text{CHOL}(A_{22}) \\
A_{23} &:= \text{TRIU}(A_{22})^{-1}A_{23} \\
A_{24} &:= \text{TRIU}(A_{22})^{-1}A_{24} \\
A_{33} &:= A_{33} - A_{23}^T A_{23} \\
A_{34} &:= A_{34} - A_{23}^T A_{24} \\
A_{44} &:= A_{44} - A_{24}^T A_{24}
\end{aligned}
$$

(c) 3rd iteration



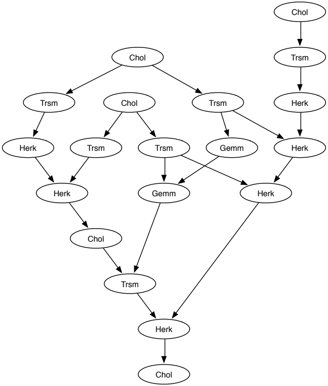
$$
\begin{aligned}
A_{33} &:= \text{CHOL}(A_{33}) \\
A_{34} &:= \text{TRIU}(A_{33})^{-1}A_{34} \\
A_{44} &:= A_{44} - A_{34}^T A_{34}
\end{aligned}
$$

(d) 4th iteration



$$
A_{44} := \text{CHOL}(A_{44})
$$

(e) 5th iteration

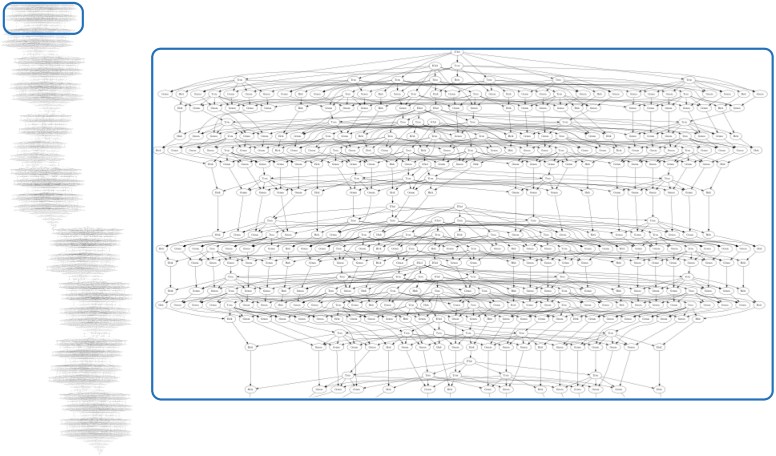


Task DAG

A sequence of tasks generated during Cholesky-by-blocks

Example from Real Problem: pwtk

- Entire task DAG is constructed to demonstrate the degree of concurrency.
- Explicit DAG is never formed and not used in task scheduling in both `Pthreads` and `Qthreads` task polices.



Numerical Examples

Test problems from UFL sparse collection

Matrix ID	# of rows(n)	# of nonzeros (nnz)	nnz/n	Description
`ecology2`	999,999	4,995,991	4.99	Circuit theory
`pwtk`	217,918	11,524,432	**52.88**	Stiffness matrix

Machine specifications

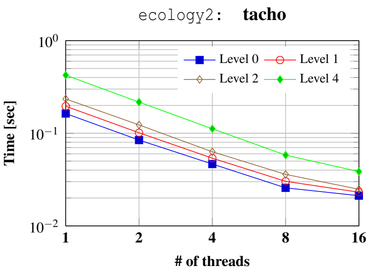
Processors	Intel Xeon E5-2670	Intel Xeon Phi	IBM Power8
# of cores	2x8	1x57	4x5
Clock speed	2.6 GHz	1.1 GHz	3.4 GHz
L2 per core	256 KB	512 KB	512 KB
L3	20 MB shared	-	8 MB per core
Compiler	Intel 15.2.164		GNU 4.9.2

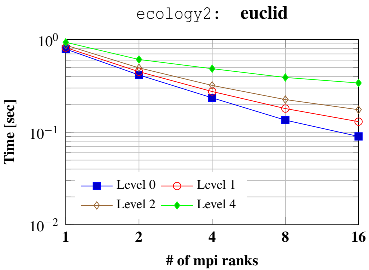
Kokkos

- `Pthreads` backend with task only interface (team size = 1).

Comparison with Euclid (Hypre)

- `Euclid` performs MPI-parallel incomplete LU.
- Parallelism is extracted from 1D rowwise partition of a matrix.
- Reverse Cuthill McKee (RCM) ordering is used to reduce the bandwidth of the matrix.
- Bandwidth of matrices increases with an increasing level of fills.

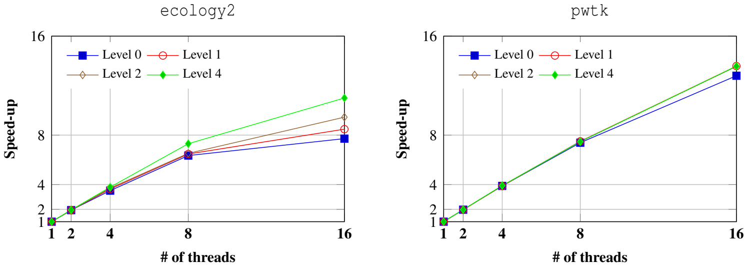




Intel Sandy Bridge

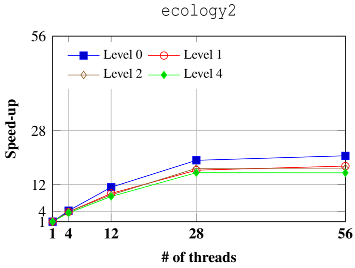
Strong Scaling : Intel Sandy Bridge

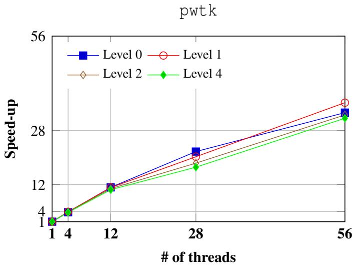
- Speed-up $= \frac{\text{Time for single-threaded Cholesky-by-blocks}}{\text{Time for parallel Cholesky-by-blocks}}$.
- Performance depends on matrix sparsity: `ecology2` is sparser and `pwtk` is denser.
- With increasing fill-level, factorization is more compute-intensive.
- Tasking overhead is constant and amortized during asynchronous parallel execution.



Strong Scaling : Intel Xeon Phi

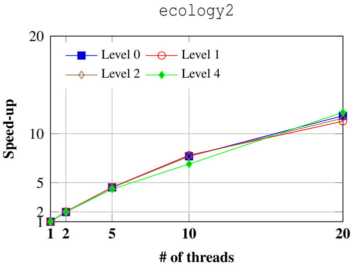
- Speed-up $= \dfrac{\text{Time for single-threaded Cholesky-by-blocks}}{\text{Time for parallel Cholesky-by-blocks}}$.
- Performance depends on matrix sparsity: `ecology2` is sparser and `pwtk` is denser.
- With increasing fill-level, factorization is more compute-intensive.
- Tasking overhead is constant and amortized during asynchronous parallel execution.

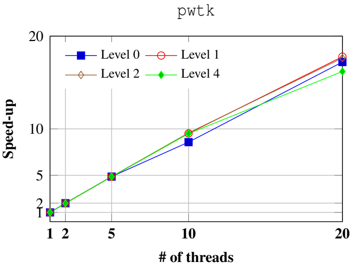




Strong Scaling : IBM Power8

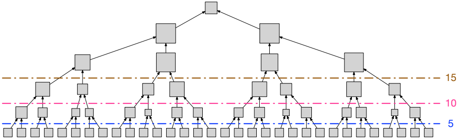
- Speed-up $= \dfrac{\text{Time for single-threaded Cholesky-by-blocks}}{\text{Time for parallel Cholesky-by-blocks}}$.
- Performance depends on matrix sparsity: `ecology2` is sparser and `pwtk` is denser.
- With increasing fill-level, factorization is more compute-intensive.
- Tasking overhead is constant and amortized during asynchronous parallel execution.



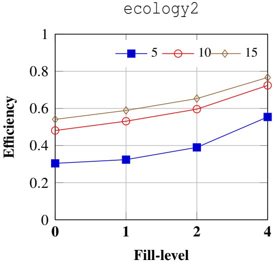


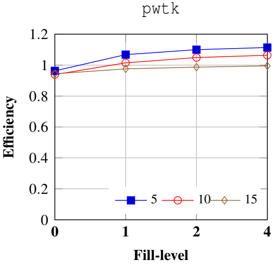
Tasking Overhead: Intel Xeon Phi

- Efficiency $= \dfrac{\text{Time for sequential Cholesky}}{\text{Time for single-threaded Cholesky-by-blocks}}$.
- With increasing fill-level, cache-friendly sparse matrix operations on blocks exploit better data locality.
- Task granularity is problem-specific on irregular problems.



	# blocks	
Treecut	`ecology2`	`pwtk`
15	83	17
10	3,071	905
5	105,864	33,182





Summary

Kokkos hybrid task-data programming model

- Presented abstractions for task-data parallelism.
- Developed dependence driven task model.
- Harnessed two tasking backends: `Pthreads` and `Qthreads`.
- `https://github.com/kokkos/kokkos`

Task-data parallel sparse matrix factorization

- Presented sparse algorithm-by-blocks for task parallel Cholesky (in)complete factorization.
- As mini-app, provided support and feedback to design task-data interface.
- Demonstrated portable performance on multicore and manycore architectures.
- `Trilinos/shylu/tacho`

Kokkos tasking API and Cholesky miniapp are in the experimental phase

Ongoing and Future Work

Kokkos hybrid task-data programming model

- Asynchronous tasking on GPUs.

Task-data parallel sparse matrix factorization

- Performance optimization for task-data hybrid parallelism:
  e.g., algorithm design and thread team overhead.
- Supernodal direct factorization: Cholesky and LDL.
- Leverage to domain decomposition FE solver in collaboration with Clark Dohrmann (1542).