A DAG-based sparse Cholesky solver for multicore architectures

Jonathan Hogg
John Reid
Jennifer Scott

CSC09 Monterey Bay October 2009
How to efficiently solve $Ax = b$ on **multicore** machines

- Introduction
- Dense systems
- Sparse systems
- Future directions and conclusions

**Today** $A$ is positive definite.
Haven’t we been solving linear systems in parallel for years? Yes — large problems on distributed memory machines

We want to solve

- Medium and large problems (more than $10^{10}$ flops)
- On desktop machines
- Shared memory, complex cache-based architectures
- 2–8 cores now in all new machines.
- Soon 16–64 cores will be standard.

Traditional MPI methods work, but can we do better?
I have an 8-core machine...

...I want to go (nearly) 8 times faster
The dense problem

Solve

\[ Ax = b \]

with \( A \)

- Symmetric and dense
- Positive definite (indefinite problems require pivoting)
- Not small (order at least a few hundred)
Pen and paper approach

Factorize $A = LL^T$ then solve $Ax = b$ as

$$Ly = b$$

$$L^T x = y$$
Pen and paper approach

Factorize $A = LL^T$ then solve $Ax = b$ as

$$Ly = b$$

$$L^T x = y$$

Algorithm:

- For each column $k$:
  - $L_{kk} = \sqrt{A_{kk}}$ (Calculate diagonal element)
  - For rows $i > k$: $L_{ik} = A_{ik} L_{kk}^{-1}$ (Divide column by diagonal)
  - Update trailing submatrix
    $$A((k+1:n)(k+1:n) \leftarrow A((k+1:n)(k+1:n) - L(k+1:n)k L_{(k+1:n)k}^T$$
Serial approach

Exploit caches
Use algorithm by blocks

- Same algorithm, but submatrices not elements
- $10 \times$ faster than a naive implementation
- Built using Basic Linear Algebra Subroutines (BLAS)
Factorize diagonal

Factor(col)
Cholesky by blocks

Solve column block

Solve(row, col)
Cholesky by blocks

Update block

Update(row, source col, target col)
Parallelism mechanisms

MPI  Designed for distributed memory, requires substantial changes
OpenMP  Designed for shared memory
pthreads  POSIX threads, no Fortran API
ITBB  Intel Thread Building Blocks, no Fortran API
Coarrays  Not yet widely supported
Parallelism mechanisms

- **MPI**  Designed for distributed memory, requires substantial changes
- **OpenMP**  Designed for shared memory
- **pthreads**  POSIX threads, no Fortran API
- **ITBB**  Intel Thread Building Blocks, no Fortran API
- **Coarrays**  Not yet widely supported
Parallelism mechanisms

**MPI**  Designed for distributed memory, requires substantial changes

**OpenMP**  Designed for shared memory

**pthreads**  POSIX threads, no Fortran API

**ITBB**  Intel Thread Building Blocks, no Fortran API

**Coarrays**  Not yet widely supported
Traditional approach

Just parallelise the operations

*Solve*(row,col)  Can do the solve in parallel

*Update*(row,scol,tcol)  Easily split as well
Traditional approach

Just parallelise the operations

**Solve**(row,col) Can do the solve in parallel

**Update**(row,scol,tcol) Easily split as well

What does this look like...
Parallel right looking
What do we really need to synchronise?
What do we really need to synchronise?

Represent each block operation (Factor, Solve, Update) as a task.

Tasks have dependencies.
What do we really need to synchronise?

Represent each block operation (Factor, Solve, Update) as a task.

Tasks have dependencies.

Represent this as a directed graph

- Tasks are vertices
- Dependencies are directed edges

It is acyclic — hence have a Directed Acyclic Graph (DAG).
What do we really need to synchronise?

Represent each block operation (Factor, Solve, Update) as a task.

Tasks have dependencies.

Represent this as a directed graph

- Tasks are vertices
- Dependencies are directed edges

It is acyclic — hence have a Directed Acyclic Graph (DAG).

Approach used by Buttari, Dongarra, Kurzak, Langou, Luszczek, Tomov (2006)
A DAG-based sparse Cholesky solver

**Factor(col)** \( A_{kk} = L_{kk}L_{kk}^T \)

**Solve(row, col)** \( L_{ik} = A_{ik}L_{kk}^{-T} \)

**Update(row, scol, tcol)** \( A_{ij} \leftarrow A_{ij} - L_{ik}L_{jk}^T \)
**Task DAG**

**Factor (col)**

\[ A_{kk} = L_{kk}L_{kk}^T \]

**Solve (row, col)**

\[ L_{ik} = A_{ik}L_{kk}^{-T} \]

**Update (row, scol, tcol)**

\[ A_{ij} \leftarrow A_{ij} - L_{ik}L_{jk}^T \]

A DAG-based sparse Cholesky solver
J.D. Hogg and J.K. Reid and J.A. Scott

A DAG-based sparse Cholesky solver
Performance using 8 threads (dgemm peak is 72.8 Gflop/s)
## Speedup for dense case

<table>
<thead>
<tr>
<th>( n )</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>3.2</td>
</tr>
<tr>
<td>2500</td>
<td>5.7</td>
</tr>
<tr>
<td>10000</td>
<td>7.2</td>
</tr>
<tr>
<td>20000</td>
<td>7.4</td>
</tr>
</tbody>
</table>
### Speedup for dense case

<table>
<thead>
<tr>
<th>$n$</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>3.2</td>
</tr>
<tr>
<td>2500</td>
<td>5.7</td>
</tr>
<tr>
<td>10000</td>
<td>7.2</td>
</tr>
<tr>
<td>20000</td>
<td>7.4</td>
</tr>
</tbody>
</table>

New dense DAG code `HSL_MP54` available in HSL2007.
Sparse case?

So far, so dense. What about sparse factorizations?
Sparse matrices

- Sparse matrix is mostly zero — only track non-zeros.
- Factor $L$ is denser than $A$.
- Extra entries are known as fill-in.
- Reduce fill-in by preordering $A$. 

J.D.Hogg and J.K. Reid and J.A.Scott

A DAG-based sparse Cholesky solver
Direct methods

Generally comprise four phases:

- **Reorder** Symmetric permutation $P$ to reduce fill-in.
- **Analyse** Predict non-zero pattern. Build elimination tree.
- **Factorize** Using data structures built in analyse phase, perform the numerical factorization.
- **Solve** Using computed factors solve $Ax = b$.

Aim: Organise computations to use dense kernels on submatrices.
Elimination and assembly tree

The **elimination tree** provides partial ordering of the operations.

If $U$ is a descendant of $V$, we must factorize $U$ first.

To exploit BLAS, combine adjacent nodes whose cols have same (or similar) sparsity structure.

Condensed tree is **assembly tree**.
Existing parallel approaches usually rely on two levels of parallelism.

**Tree-level parallelism:** assembly tree specifies only partial ordering (parent processed after its children). Independent subtrees processed in parallel.

**Node-level parallelism:** parallelism within operations at a node. Normally used near the root.
Existing parallel approaches usually rely on two levels of parallelism

**Tree-level parallelism:** assembly tree specifies only partial ordering (parent processed after its children). Independent subtrees processed in parallel.

**Node-level parallelism:** parallelism within operations at a node. Normally used near the root.

**Our experience:** speedups less than ideal on multicore machines.
Basic idea: Extend DAG-based approach to the sparse case by adding new type of task to perform sparse update operations.
**Basic idea:** Extend DAG-based approach to the sparse case by adding new type of task to perform sparse update operations.

Hold set of contiguous cols of $L$ with (nearly) same pattern as a dense trapezoidal matrix, referred to as **nodal matrix**.
**Basic idea:** Extend DAG-based approach to the sparse case by adding new type of task to perform sparse update operations.

Hold set of contiguous cols of $L$ with (nearly) same pattern as a dense trapezoidal matrix, referred to as **nodal matrix**.

Divide the nodal matrix into blocks and perform tasks on the blocks.
factorize(diag) Computes dense Cholesky factor $L_{\text{triang}}$ of the triangular part of block diag on diagonal. If block trapezoidal, perform triangular solve of rectangular part

$$L_{\text{rect}} \leftarrow L_{\text{rect}} L_{\text{triang}}^{-T}$$
factorize(diag) Computes dense Cholesky factor $L_{\text{triang}}$ of the triangular part of block diag on diagonal. If block trapezoidal, perform triangular solve of rectangular part

$$L_{\text{rect}} \leftarrow L_{\text{rect}}L_{\text{triang}}^{-T}$$

solve(dest, diag) Performs triangular solve of off-diagonal block dest by Cholesky factor $L_{\text{triang}}$ of block diag on its diagonal.

$$L_{\text{dest}} \leftarrow L_{\text{dest}}L_{\text{triang}}^{-T}$$
Tasks in sparse DAG

`update_internal(dest, rsrc, csrc)`

Within nodal matrix, performs update

\[
L_{\text{dest}} \leftarrow L_{\text{dest}} - L_{\text{rsrc}} L_{\text{csrc}}^T
\]
Tasks in sparse DAG

update_between(dest, snode, scol)
Performs update

\[ L_{dest} \leftarrow L_{dest} - L_{rsr}L_{csr}^T \]

- where \( L_{dest} \) is a submatrix of the block \( dest \) of an ancestor of node \( snode \)
- \( L_{rsr} \) and \( L_{csr} \) are submatrices of contiguous rows of block column \( scol \) of \( snode \).
update_between(dest, snode, scol)

1. Form outer product $L_{rsr}L_{csrc}^T$ into Buffer.
2. Distribute the results into the destination block $L_{dest}$. 
During analyse, calculate number of tasks to be performed for each block of $L$. 
During analyse, calculate number of tasks to be performed for each block of $L$.

During factorization, keep running count of outstanding tasks for each block.
During analyse, calculate number of tasks to be performed for each block of $L$.

During factorization, keep running count of outstanding tasks for each block.

When count reaches 0 for block on the diagonal, store factorize task and decrement count for each off-diagonal block in its block column by one.
Dependency count

During analyse, calculate number of tasks to be performed for each block of $L$.

During factorization, keep running count of outstanding tasks for each block.

When count reaches 0 for block on the diagonal, store factorize task and decrement count for each off-diagonal block in its block column by one.

When count reaches 0 for off-diagonal block, store solve task and decrement count for blocks awaiting the solve by one. Update tasks may then be spawned.
Each cache keeps small stack of tasks that are intended for use by threads sharing this cache.

Tasks added to or drawn from top of local stack. If becomes full, move bottom half to task pool.

Tasks in pool given priorities:

1. **factorize**    Highest priority
2. **solve**
3. **update_internal**
4. **update_between**  Lowest priority
## Sparse DAG results

Results on machine with 2 Intel E5420 quad core processors.

<table>
<thead>
<tr>
<th>Problem</th>
<th>cores</th>
<th>Time 1</th>
<th>Time 8</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>DNVS/thread</td>
<td>5.25</td>
<td>0.98</td>
<td>5.36</td>
<td></td>
</tr>
<tr>
<td>GHS_psdef/apache2</td>
<td>30.1</td>
<td>5.07</td>
<td>5.94</td>
<td></td>
</tr>
<tr>
<td>Koutsovasilis/F1</td>
<td>37.8</td>
<td>6.05</td>
<td>6.24</td>
<td></td>
</tr>
<tr>
<td>JGD_Trefethen/Trefethen_20000b</td>
<td>102</td>
<td>16.5</td>
<td>6.18</td>
<td></td>
</tr>
<tr>
<td>ND/nd24k</td>
<td>335</td>
<td>53.7</td>
<td>6.23</td>
<td></td>
</tr>
</tbody>
</table>

J.D.Hogg and J.K. Reid and J.A.Scott  
A DAG-based sparse Cholesky solver
Comparisons with other solvers, one thread

![Graph showing comparisons between solvers over problem index](image-url)
Comparisons with other solvers, 8 threads

![Graph showing time comparison between MUMPS and PARDISO]
Sparse DAG approach very encouraging for multicore architectures.
Indefinite case

Sparse DAG approach very encouraging for multicore architectures.

BUT

- Results reported so far, only for positive definite case.
- Indefinite case is harder because of pivoting.
- We use block column dependency counts and combine factor and solve tasks.
- Preliminary results: speed ups not quite so good.
Code availability

New sparse DAG code is HSL_MA87.

To be included within HSL.

If you want to try it out, let us know.