A DAG-based sparse Cholesky solver for multicore architectures

Jennifer Scott
Jonathan Hogg
John Reid

Rutherford Appleton Laboratory

Birmingham March 2010
How to efficiently solve $Ax = b$ on *multicore* machines

- Introduction
- Dense systems
- Sparse systems
- Conclusions
What’s the problem?

We wish to solve

\[ Ax = b \]

where \( A \) is

LARGE

sparse

What is sparse? \( A \) is sparse if

- many entries are zero
- it is worthwhile to exploit these zeros.
Solving sparse linear systems

Two main classes of methods:

- **Direct methods** are usually variants of Gaussian elimination and involve explicit factorization eg $PAQ = LU$
  - $L$, $U$ lower and upper triangular matrices
  - $P$, $Q$ are permutation matrices
  - Solution process completed by (easy) triangular solves $Ly = Pb$ and $Uz = y$ then $x = Qz$

- **Iterative methods** eg conjugate gradients, GMRES, BiCGSTAB, MINRES ...

Jennifer Scott
A DAG-based sparse Cholesky solver
Direct Methods

Advantages:
- High accuracy
- Robust. Can be used as black box solvers
- Solving for multiple right-hand sides cheap

Disadvantages:
- Memory required grows more rapidly than problem size
- Difficult to code efficiently
Iterative methods

Advantages

- need only a small number of arrays of length $n$
- easy to code
- speed depends on matrix-vector products ... parallelise
- can choose accuracy

Disadvantages

- Lack of robustness
- Require preconditioner but how to choose?
- May want to solve for many right hand sides.
What direct solvers are there?

- Developed since early 1970s
- Significant work into development of serial codes.
- Well-known HSL packages include MA27, MA57, MA48 ...
- Some codes for distributed memory machines (MPI)
  SuperLU, MUMPS ...
- Other codes developed for shared memory machines
  (OpenMP)
  PARDISO, PASTIX, WSMP ...

Now we have the **multicore challenge** ... existing codes do not exploit new architecture
What’s the challenge?

We want to solve

- Medium and large linear systems (more than $10^{10}$ flops)
- On desktop machines
- Shared memory, complex cache-based architectures
- 2–6 cores now in all new machines.
- Soon 16–64 cores will be standard.
Why is multicore hard?

In the beginning...

CPU

Main Memory
Why is multicore hard?

Add cache...
Why is multicore hard?

Go parallel...

Processor 0

Core

L2 Cache

Processor 1

Core

L2 Cache

Main Memory
Why is multicore hard?

Multicore...

Processor 0
Package 0
Core 0
L2 Cache
Core 1

Package 1
Core 2
L2 Cache
Core 3

Processor 1
Package 2
Core 4
L2 Cache
Core 5

Package 3
L2 Cache
Core 6
Core 7

Main Memory
Tackling multicore

How do we get good performance on multicore?

- **Parallelism**: Obviously...
- **Data reuse**: Shared caches
- **Cache locality**: Hard with *sparse matrices*
- **Experimentation**: Profiling, trying things out
And so ...

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

Aim to exploit caches Work with blocks

- Same algorithm, but **submatrices not elements**
  - Factor: \( A_k = L_{kk}L_{kk}^T \)
  - Solve: \( L_{ik} = A_{ik}L_{kk}^{-1} \)
  - Update: \( A_{ij} \leftarrow A_{ij} - L_{ik}L_{kj}^T \)

- 10× faster than a naive implementation
- Built using Level 3 Basic Linear Algebra Subroutines (BLAS)
  eg gemm for the update operations
Cholesky by blocks

Factorize diagonal

Factor(col)
Solve column block

Solve(row, col)
Update block

Update(row, source col, target col)
Traditional approach

Just parallelise the operations

 Solve\((\text{row, col})\)  Can do the solve in parallel
 Update\((\text{row, scol, tcol})\)  Easily split as well
Traditional approach

Just parallelise the operations

\textbf{Solve}(\textit{row,}\textit{col}) Can do the solve in parallel

\textbf{Update}(\textit{row,}\textit{scol,}\textit{tcol}) Easily split as well

What does this look like...
Parallel right looking
Aims for multicore algorithm

Low granularity  Many many more tasks than cores.
Asyncronicity  Don’t use tight coupling, only enforce necessary ordering.
Dynamic Scheduling  Static (precomputed) scheduling is easily upset.
Locality of reference  Cache/performance ratios will only get worse, try not to upset them.
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.

Factor must wait for fully updated $A_{kk}$.

Solve must wait for fully updated $A_{ik}$ and for $L_{kk}$.

Update must wait for $L_{ik}$ and $L_{jk}$. 
What do we **really** need to synchronise?

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

Tasks have **dependencies**.

Factor must wait for fully updated $A_{kk}$.

Solve must wait for fully updated $A_{ik}$ and for $L_{kk}$.

Update must wait for $L_{ik}$ and $L_{jk}$.

Represent this as a directed graph

- Tasks are vertices
- Dependencies are directed edges
What do we really need to synchronise?

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

Tasks have dependencies.

Factor must wait for fully updated $A_{kk}$.

Solve must wait for fully updated $A_{ik}$ and for $L_{kk}$.

Update must wait for $L_{ik}$ and $L_{jk}$.

Represent this as a directed graph

- Tasks are vertices
- Dependencies are directed edges

It is acyclic — hence have a Directed Acyclic Graph (DAG)
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) \( A_{ij} \leftarrow A_{ij} - L_{ik} L_{jk}^T \)
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) $A_{ij} \leftarrow A_{ij} - L_{ik}L_{jk}^T$
Main Loop

```
loop
  get_task()
  do_task()
  reduce_dep()
  add_tasks()
end loop
```
Main Loop

```plaintext
loop
  get_task()
  do_task()
  reduce_dep()
  add_tasks()
end loop

Different threads accessing the same memory location simultaneously = bad!
Need to be careful — use locks.
```
Performance in Gfop/s for dense case

8 core machine, peak performance of gemm is 72.8

<table>
<thead>
<tr>
<th>threads</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>n = 500</td>
<td>5.6</td>
<td>8.6</td>
<td>13.4</td>
<td>17.7</td>
<td>3.2</td>
</tr>
<tr>
<td>2500</td>
<td>7.6</td>
<td>14.5</td>
<td>26.9</td>
<td>43.5</td>
<td>5.7</td>
</tr>
<tr>
<td>10000</td>
<td>8.6</td>
<td>17.1</td>
<td>33.6</td>
<td>61.9</td>
<td>7.2</td>
</tr>
<tr>
<td>20000</td>
<td>8.8</td>
<td>17.7</td>
<td>35.1</td>
<td>65.5</td>
<td>7.4</td>
</tr>
</tbody>
</table>
8 core machine, peak performance of gemm is 72.8

<table>
<thead>
<tr>
<th>threads</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n = 500$</td>
<td>5.6</td>
<td>8.6</td>
<td>13.4</td>
<td>17.7</td>
<td>3.2</td>
</tr>
<tr>
<td>2500</td>
<td>7.6</td>
<td>14.5</td>
<td>26.9</td>
<td>43.5</td>
<td>5.7</td>
</tr>
<tr>
<td>10000</td>
<td>8.6</td>
<td>17.1</td>
<td>33.6</td>
<td>61.9</td>
<td>7.2</td>
</tr>
<tr>
<td>20000</td>
<td>8.8</td>
<td>17.7</td>
<td>35.1</td>
<td>65.5</td>
<td>7.4</td>
</tr>
</tbody>
</table>

Dense DAG-based Cholesky solver HSL_MP54 available in HSL 2007.
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.
- Limit fill-in by preordering $A$. 
Direct methods

Generally comprise four phases:

- **Reorder** Symmetric permutation $P$ to limit fill-in (nested dissection, minimum degree ...)
- **Analyse** Predict non-zero pattern of $L$ and build data structures for this.
- **Factorize** Use these data structures to perform factorization.
- **Solve** Use factor to solve $Ax = b$.

Aim: Organise computation to use dense kernels on submatrices.
Data structures

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

Divide nodal matrix into blocks and perform tasks on blocks.
Nodal matrices

Assuming null rows have been removed, a block column of $L$ is stored as a dense submatrix.

- Sparse $L$ is made up of many of these dense block columns
- For each nodal matrix, hold integer list of rows involved
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.

Tasks in sparse DAG:

factorize Computes dense Cholesky factor $L_{\text{diag}}$ of the block diag on diagonal. As in dense case, use LAPACK code.
Basic idea: Extend DAG-based approach to the sparse case by adding new type of task to perform sparse update operations.

Tasks in sparse DAG:

factorize Computes dense Cholesky factor $L_{\text{diag}}$ of the block $\text{diag}$ on diagonal. As in dense case, use LAPACK code.

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

$$L_{\text{dest}} \leftarrow L_{\text{dest}} L^{-T}_{\text{diag}}$$

Again, as in dense case, use BLAS 3 trsm
update\_internal
Within nodal matrix, performs update

\[ L_{\text{dest}} \leftarrow L_{\text{dest}} - L_{\text{rsrc}} L_{\text{csrc}}^T \]

This is like in dense case and we can use BLAS 3 gemm
update\_between
Performs update

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

where \( L_{\text{dest}} \) belongs to one nodal matrix and \( L_{\text{rsr}} \) and \( L_{\text{csr}} \) belong to another.
update_between

1. Form outer product $L_{rsrcl} L_{csrcc}^T$ 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 $L$. 
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.
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
The ratios of HSL_MA87 factorize times on 2, 4 and 8 cores to its factorize time on a single core.
Comparisons with other solvers, one thread

The ratios of the PARDISO, TAUCS and PaStiX factorize times to the HSL_MA87 factorize time

![Graph showing the ratios of factorization times for different solvers]
Comparisons with other solvers, 8 threads

The ratios of the PARDISO, TAUCS and PaStiX factorize times to the HSL_MA87 factorize time

![Chart showing the ratios of different solvers to HSL_MA87 factorization times](chart.png)
Indefinite case

Sparse DAG approach very encouraging for our 8 core machine
Indefinite case

Sparse DAG approach very encouraging for our 8 core machine

BUT

- So far, only considered positive definite case.
- Indefinite case is harder because of need for pivoting. Saddle-point systems are common.

\[
\begin{pmatrix}
H & A^T \\
A & 0 \\
\end{pmatrix}
\begin{pmatrix}
x \\
y \\
\end{pmatrix}
=
\begin{pmatrix}
a \\
b \\
\end{pmatrix}
\]
Indefinite case

What changes do we need?

- We combine factor task and solve tasks and hold block column dependency counts.
- A block column can only be factorized once its dependency count reaches zero.
- Factorization of block column incorporates threshold pivoting. Chooses $1 \times 1$ and $2 \times 2$ pivots so the computed factorization is

$$A = (PL)D(LP)^T$$

with $D$ block diagonal.
Complications in indefinite case

- Pivoting adds overhead
- Less scope for parallelism (the factorize-solve tasks are large towards end of factorization)
- Data structures from analyse have to be modified to accommodate delayed pivots (those that fail stability test)
- Code is even more complicated!
## Indefinite results

Good results on large problems

<table>
<thead>
<tr>
<th></th>
<th>threads</th>
<th>1</th>
<th>8</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oberwolfach/t3dh</td>
<td>(79,171)</td>
<td>12.1</td>
<td>2.24</td>
<td>5.9</td>
</tr>
<tr>
<td>ND/nd12k</td>
<td>(36,000)</td>
<td>88.5</td>
<td>15.2</td>
<td>5.8</td>
</tr>
<tr>
<td>GHS_indef/sparsine</td>
<td>(50,000)</td>
<td>250</td>
<td>44.4</td>
<td>5.6</td>
</tr>
<tr>
<td>PARSEC/GaAsH6</td>
<td>(61,349)</td>
<td>264</td>
<td>45.2</td>
<td>5.8</td>
</tr>
</tbody>
</table>
Concluding remarks and open questions

- Our code HSL_MA87 is performing well on our 8 core machine.
- Number of cores will increase in future. How well will this approach scale?
- Can we improve performance for indefinite case? (new orderings, new pivoting strategies ...)
- Reported results are factorize times. Times for forward/back substitutions do not give such good speedups (2 is typical). Problem is memory traffic. How to tackle this?
New sparse DAG code is HSL\_MA87. Fortran 2003.

Will shortly be available as part of HSL.

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