A DAG-based sparse Cholesky solver for multicore architectures

Jennifer Scott
Jonathan Hogg
John Reid

Rutherford Appleton Laboratory

Birmingham March 2010

Outline of talk

How to efficiently solve $A\mathbf{x} = \mathbf{b}$ on multicore machines

- Introduction
- Dense systems
- Sparse systems
- Conclusions

What's the problem?

We wish to solve

$$A\mathbf{x} = \mathbf{b}$$

where A is

LARGE

s p a r s e

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 $L\mathbf{y} = P\mathbf{b}$ and $U\mathbf{z} = \mathbf{y}$ then $\mathbf{x} = Q\mathbf{z}$
- **Iterative methods** eg conjugate gradients, GMRES, BiCGSTAB, MINRES ...

Direct Methods

Advantages:

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

Disdvantages:

- 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

Disdvantages

- Lack of robustness
- Require preconditioner but how to choose?
  Highly problem dependent. Difficult in parallel.
- 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

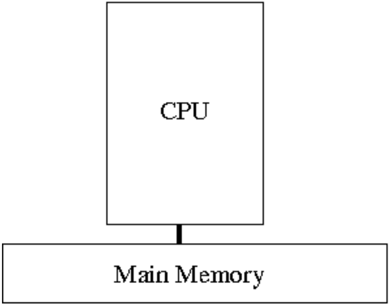
What's the challenge?

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.

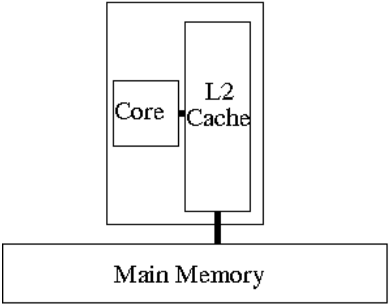
Why is multicore hard?

In the beginning...



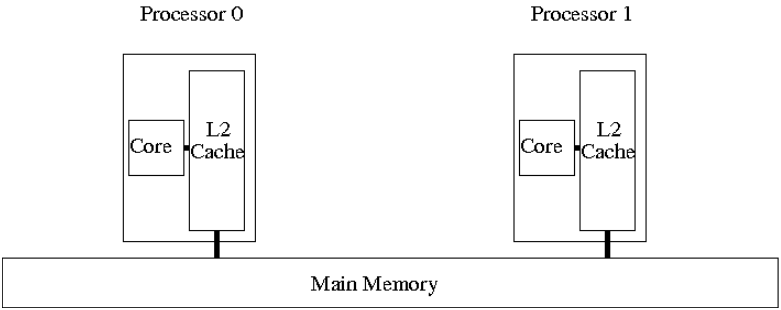
Why is multicore hard?

Add cache...



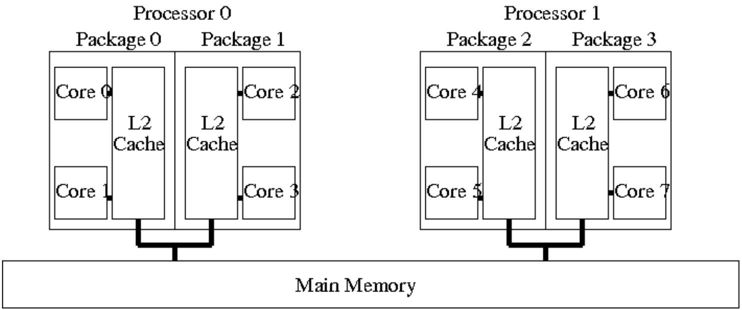
Why is multicore hard?

Go parallel...



Why is multicore hard?

Multicore...



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

$$A\mathbf{x} = \mathbf{b}$$

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 $A\mathbf{x} = \mathbf{b}$ as

$$\begin{aligned} L\mathbf{y} &= \mathbf{b} \\ L^T\mathbf{x} &= \mathbf{y} \end{aligned}$$

Pen and paper approach

Factorize $A = LL^T$ then solve $A\mathbf{x} = \mathbf{b}$ as

$$\begin{aligned} L\mathbf{y} &= \mathbf{b} \\ L^T\mathbf{x} &= \mathbf{y} \end{aligned}$$

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 $A_{(k+1:n)(k+1:n)} \leftarrow A_{(k+1:n)(k+1:n)} - L_{(k+1:n)k}L_{(k+1:n)k}^T$

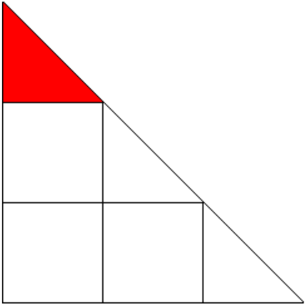
Serial approach

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: $A_{ij} \leftarrow A_{ij} - L_{ik} L_{kj}^T$
- $10\times$ faster than a naive implementation
- Built using Level 3 Basic Linear Algebra Subroutines (BLAS) eg gemm for the update operations

Cholesky by blocks

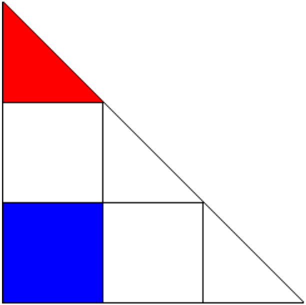
Factorize diagonal



Factor(col)

Cholesky by blocks

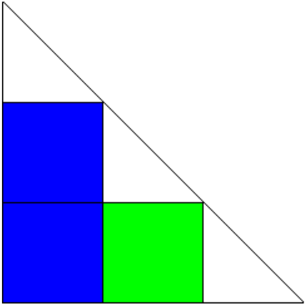
Solve column block



Solve(row, col)

Cholesky by blocks

Update block



Update(row, source col, target col)

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

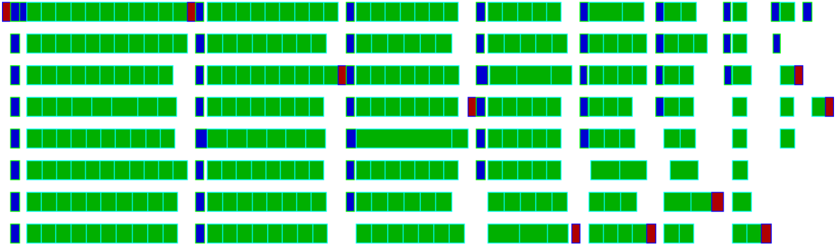
Just parallelise the operations

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



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.

DAGs

What do we really need to synchronise?

DAGs

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}.

DAGs

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

DAGs

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}.

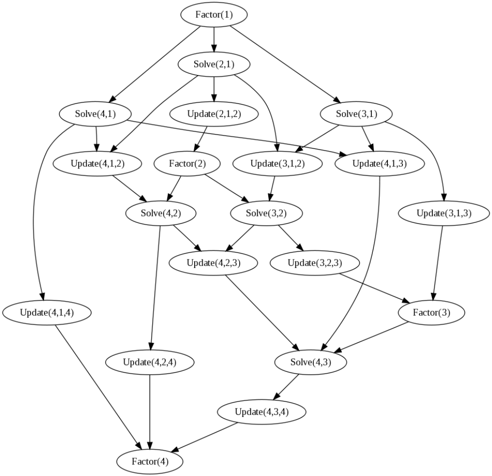
Update must wait for L_{ik} and L_{jk}.

Represent this as a directed graph

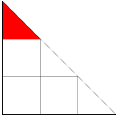
- Tasks are vertices
- Dependencies are directed edges

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

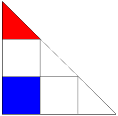
Task DAG



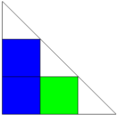
Factor(col) $A_{kk} = L_{kk} L_{kk}^T$



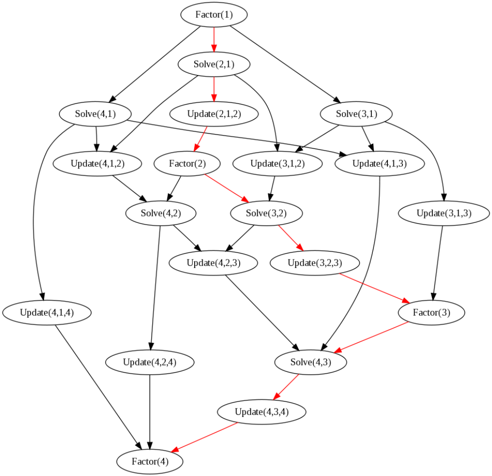
Solve(row, col) $L_{ik} = A_{ik} L_{kk}^{-T}$



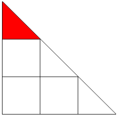
Update(row, scol, tcol)
$A_{ij} \leftarrow A_{ij} - L_{ik} L_{jk}^T$



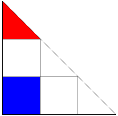
Task DAG



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

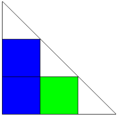


Solve(row, col) $L_{ik} = A_{ik} L_{kk}^{-T}$



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

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

threads	1	2	4	8	speedup
$n = 500$	5.6	8.6	13.4	17.7	3.2
2500	7.6	14.5	26.9	43.5	5.7
10000	8.6	17.1	33.6	61.9	7.2
20000	8.8	17.7	35.1	65.5	7.4

Performance in Gfop/s for dense case

8 core machine, peak performance of gemm is 72.8

threads	1	2	4	8	speedup
$n = 500$	5.6	8.6	13.4	17.7	3.2
2500	7.6	14.5	26.9	43.5	5.7
10000	8.6	17.1	33.6	61.9	7.2
20000	8.8	17.7	35.1	65.5	7.4

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 $A\mathbf{x} = \mathbf{b}$.

Aim: Organise computation to use dense kernels on submatrices.

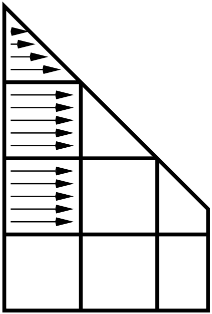
Data structures

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

Sparse DAG

Basic idea: Extend DAG-based approach to the sparse case by adding new type of task to perform sparse update operations.

Sparse DAG

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_{diag} of the block `diag` on diagonal. As in defnse case, use LAPACK code.

Sparse DAG

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_{diag} of the block `diag` on diagonal. As in defnse case, use LAPACK code.

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

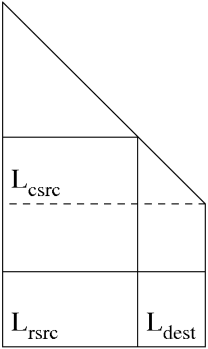
$$L_{dest} \Leftarrow L_{dest} L_{diag}^{-T}$$

Again, as in dense case, use BLAS 3 `trsm`

Tasks in sparse DAG

update_internal

Within nodal matrix, performs update



$$L_{dest} \Leftarrow L_{dest} - L_{rsrc} L_{csrc}^T$$

This is like in dense case and we can use BLAS 3 `gemm`

Tasks in sparse DAG

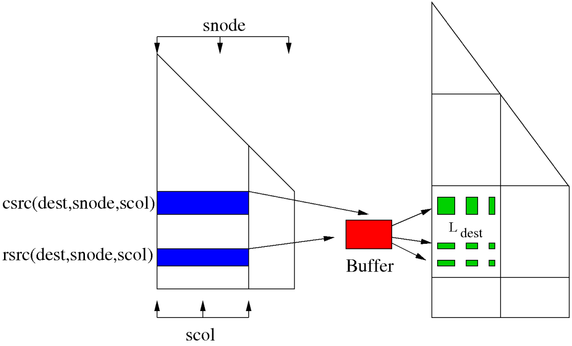
update_between

Performs update

$$L_{dest} \Leftarrow L_{dest} - L_{rsrc} L_{csrc}^T$$

where L_{dest} belongs to one nodal matrix and L_{rsrc} and L_{csrc} belong to another.

update_between



1. Form outer product $L_{rsrc}L_{csrc}^T$ into Buffer.
2. Distribute the results into the destination block L_{dest}.

Dependency count

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.

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.

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.

Task pool

Each cache keeps small stack of tasks that are intended for use by threads sharing this cache.

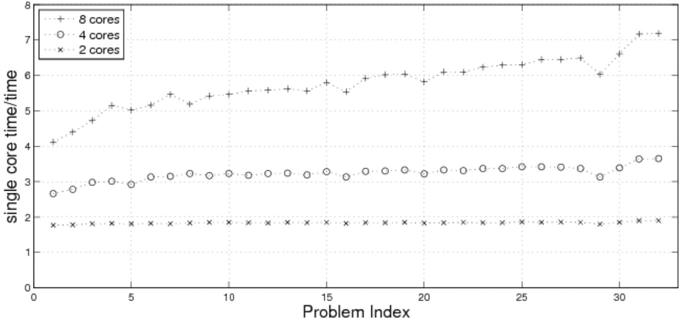
Tasks added to or drawn from top of local stack. If becomes full, move bottom half to task pool.

Tasks in pool given priorities:

1.	**factorize**	Highest priority
2.	**solve**	
3.	**update_internal**	
4.	**update_between**	Lowest priority

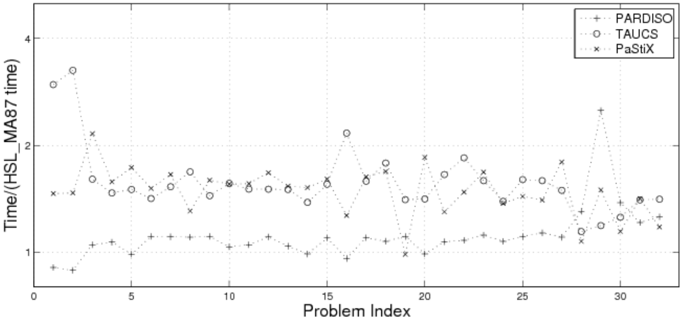
Sparse DAG results

The ratios of HSL_MA87 factorize times on 2, 4 and 8 cores to its factorize time on a single core.



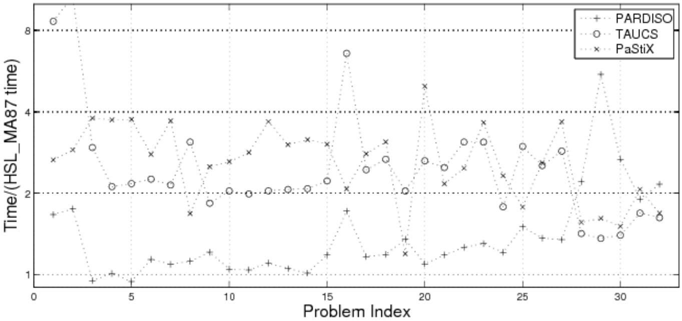
Comparisons with other solvers, one thread

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



Comparisons with other solvers, 8 threads

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



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 1×1 and 2×2 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

	threads	1	8	speedup
Oberwolfach/t3dh	(79,171)	12.1	2.24	5.9
ND/nd12k	(36,000)	88.5	15.2	5.8
GHS_indef/sparsine	(50,000)	250	44.4	5.6
PARSEC/GaAsH6	(61,349)	264	45.2	5.8

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 subsitutions do not give such good speedups (2 is typical). Problem is memory traffic. How to tackle this?

Code availability

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.