

# The multicore challenge: the sparse indefinite case

**Jonathan Hogg**  
**Jennifer Scott**

Sparse Days at CERFACS, 15 June 2010

# Outline of talk

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

( $A$  symmetric)

- Dense positive-definite systems
- Large sparse **positive-definite** systems
- Large sparse **indefinite** systems

# Dense positive-definite systems

Want to factorize  $A = LL^T$

Simple block algorithm:

For  $k = 1, 2, \dots$ :

- $A_{kk} = L_{kk}L_{kk}^T$  (Factor)
- For  $i > k$ :  $L_{ik} = A_{ik}L_{kk}^{-T}$  (Triangular Solve)
- For  $i, j > k$ :  $A_{ij} \leftarrow A_{ij} - L_{ik}L_{jk}^T$  (Update)

# Dense positive-definite systems

Want to factorize  $A = LL^T$

Simple block algorithm:

For  $k = 1, 2, \dots$ :

- $A_{kk} = L_{kk}L_{kk}^T$  (Factor)
- For  $i > k$ :  $L_{ik} = A_{ik}L_{kk}^{-T}$  (Triangular Solve)
- For  $i, j > k$ :  $A_{ij} \leftarrow A_{ij} - L_{ik}L_{jk}^T$  (Update)

Note:

- Allows use of BLAS
- Considerable freedom of order of operations
- Want to order operations to maximise parallel performance

# DAGs

What do we **really** need to synchronise?

# DAGs

What do we **really** need to synchronise?

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

Tasks have **dependencies**.

# DAGs

What do we **really** need to synchronise?

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

Tasks have **dependencies**.

Represent this implicitly as a directed graph

- Tasks are vertices
- Dependencies are directed edges

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

# DAGs

What do we **really** need to synchronise?

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

Tasks have **dependencies**.

Represent this implicitly as a directed graph

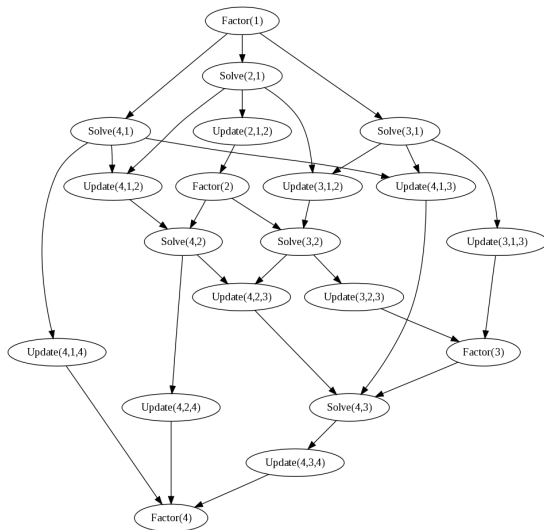
- Tasks are vertices
- Dependencies are directed edges

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

Approach used by Buttari, Dongarra, Kurzak, Langou, Luszczek, Tomov (2006)



# Task DAG (4 blocks)



## Speedup for dense case

Results on machine with 2 Intel E5420 quad core processors.

$n$	Speedup
500	3.2
2500	5.7
10000	7.2
20000	7.4

Dense DAG code HSL\_MP54 available in HSL2007.

## Sparse case?

So far, so dense. What about **sparse** factorizations?

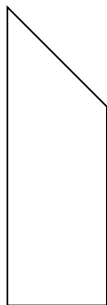
## Sparse case?

So far, so dense. What about **sparse** factorizations?

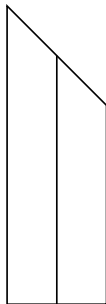
We want to generalise dense case to sparse case.

## Nodal matrix

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

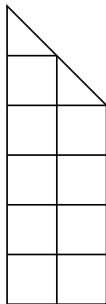


# Nodal matrix



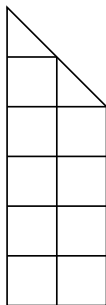
Divide nodal matrix into block columns

# Nodal matrix



Divide each block column into (square) dense blocks

# Nodal matrix



- Basic operation unit is the block.
- Tasks are performed using these blocks



## Tasks in sparse positive-definite case

Cholesky factorization of sparse  $A$  can be expressed using four types of operations on the blocks of  $L$ .

**factor\_block( $L_{diag}$ )** Computes dense Cholesky factor  $L_{diag}$  of block on diagonal.

## Tasks in sparse positive-definite case

Cholesky factorization of sparse  $A$  can be expressed using four types of operations on the blocks of  $L$ .

**factor\_block( $L_{diag}$ )** Computes dense Cholesky factor  $L_{diag}$  of block on diagonal.

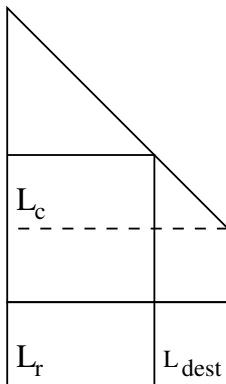
**solve\_block( $L_{dest}$ )** Performs triangular solve of off-diagonal block  $L_{dest}$  by Cholesky factor  $L_{diag}$  of block on its diagonal.

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

## Tasks in sparse positive-definite case

`update_internal( $L_{dest}$ ,  $col$ )`

Within *node*, performs update



$$L_{dest} \Leftarrow L_{dest} - L_r L_c^T$$

# Tasks in sparse DAG

`update_between( $L_{dest}$ , node, col)`

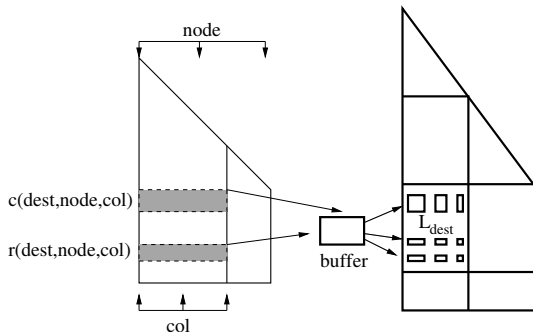
Performs update

$$L_{dest} \Leftarrow L_{dest} - L_r L_c^T$$

- $L_{dest}$  is a submatrix of an ancestor node
- $L_r$  and  $L_c$  are submatrices of contiguous rows of block column *col* of *node*.

# Tasks in sparse DAG

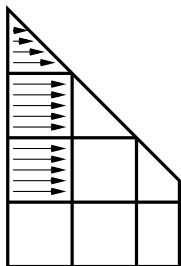
$\text{update\_between}(L_{dest}, \text{node}, \text{col})$



1. Form outer product  $L_r L_c^T$  into a buffer.
2. Distribute results into destination block  $L_{dest}$ .

## Storage of nodal matrix

- Use full storage on diagonal to allow use of efficient BLAS and LAPACK
- Store each block by rows contiguously ... removes discontinuities at row block boundaries and facilitates update tasks.



1				
4	5			
7	8	9		
10	11	12	25	
13	14	15	27	28
16	17	18	29	30
19	20	21	31	32
22	23	24	33	34

## 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, perform 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, perform 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.

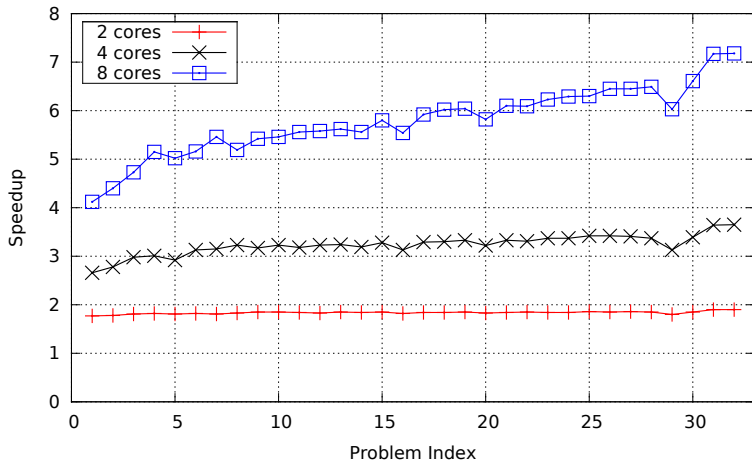
# Task pool

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.

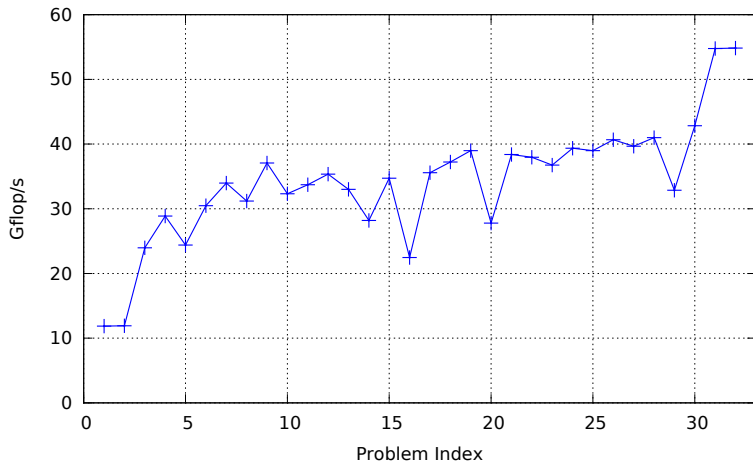
# Sparse positive-definite DAG results

Speedups for factorize phase.



## Sparse positive-definite DAG results

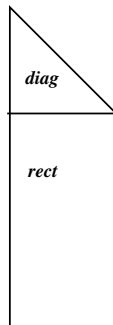
The speed of factorize phase in Gflop/s on 8 cores.  
(dgemm peak 72.8 Gflop/s)



## Sparse indefinite systems

**Extra challenge:** need to accommodate pivoting for stability

Do **not** want to restrict pivoting to within block on diagonal



Large entries in *rect* could cause problems.

# Implications

- Cannot factorize the diagonal block independently of the off-diagonal blocks.

# Implications

- Cannot factorize the diagonal block independently of the off-diagonal blocks.
- The diagonal block and the off-diagonal blocks must all have zero dependency counts.



# Implications

- Cannot factorize the diagonal block independently of the off-diagonal blocks.
- The diagonal block and the off-diagonal blocks must all have zero dependency counts.
- Necessary to combine **factor\_block** and **all solve\_block** tasks for a block column  $L_{col}$ .

# Implications

- Cannot factorize the diagonal block independently of the off-diagonal blocks.
- The diagonal block and the off-diagonal blocks must all have zero dependency counts.
- Necessary to combine **factor\_block** and **all solve\_block** tasks for a block column  $L_{col}$ .
- Separate kernel code written to perform this efficiently, incorporating threshold partial pivoting with  $1 \times 1$  and  $2 \times 2$  pivots.

# Effects

Effects of this:

- parallelism less fine-grained (large `factorize_col` task replaces smaller `factor_block` and `solve_block` tasks so we use smaller default block size)

# Effects

Effects of this:

- parallelism less fine-grained (large `factorize_col` task replaces smaller `factor_block` and `solve_block` tasks so we use smaller default block size)
- may need to expand storage determined during analyse

$$L_{col} \rightarrow L_{col}^{new}$$

( $L_{col}^{new}$  includes `delayed` columns from child nodes)

# Effects

Effects of this:

- parallelism less fine-grained (large `factorize_col` task replaces smaller `factor_block` and `solve_block` tasks so we use smaller default block size)
- may need to expand storage determined during analyse

$$L_{col} \rightarrow L_{col}^{new}$$

( $L_{col}^{new}$  includes `delayed` columns from child nodes)

- more data movement/copying

# Effects

Effects of this:

- parallelism less fine-grained (large `factorize_col` task replaces smaller `factor_block` and `solve_block` tasks so we use smaller default block size)
- may need to expand storage determined during analyse

$$L_{col} \rightarrow L_{col}^{new}$$

( $L_{col}^{new}$  includes `delayed` columns from child nodes)

- more data movement/copying
- pivot search requires access by columns (recall: block column stored by rows)

## Indefinite results for positive-definite problems

Factorize times for running positive-definite problems without and with pivoting ( $u = 0.01$ )

Problem	No pivoting		Pivoting	
	1	8	1	8
Boeing/bcsstk38	0.069	0.168	0.087	0.144
Simon/olafu	0.202	0.174	0.244	0.152
ND/nd6k	18.3	3.02	20.6	3.94
ND/nd12k	80.0	12.3	88.5	15.2

Note: smaller block used for indefinite case and this benefits smaller problems.

## Indefinite results: serial runs

Factorize times on single core. OOM indicates out of memory.

Problem	MA57	HSL_MA77	New code
Schenk_IBMNA/c-56	0.404	0.130	0.163
Simon/olafu	0.559	0.234	0.244
Koutsovasilis/F2	4.48	2.42	2.57
Cunningham/qa8fk	7.00	4.13	4.23
Oberwolfach/t3dh	20.2	11.7	12.1
Schenk_AFE/af_shell110	100	76.2	72.8
Oberwolfach/bone010	877	637	590
PARSEC/Ga41As41H72	OOM	9241	7290



## Indefinite results: good news

Factorize times on 1 and 8 cores.

Problem	1	8	speedup
Boeing/crystk03	1.29	0.36	3.58
Koutsovasilis/F2	2.57	0.57	4.51
Cunningham/qa8fk	4.23	0.88	4.79
Oberwolfach/t3dh	12.1	2.17	5.58
Schenk_AFE/af_shell10	72.8	11.7	6.22
Oberwolfach/bone010	590	88.3	6.68
PARSEC/Ga41As41H72	7290	1141	6.39

Conclude: very good results for some large problems

## Indefinite results: tough problems

Many delayed pivots cause performance hit.

Problem	num_delay	1	8	speedup
GHS_indef/sparsine	16	250	44.4	5.65
Schenk_IBMA/c-62	28728	9.07	4.93	1.84
GHS_indef/aug3d	144955	36.5	25.9	1.41

## Indefinite results: possible simple remedies

- Use relaxed pivoting (allow partial pivoting threshold parameter to reduce so that potentially less stable pivots are chosen).
- Static pivoting (no delays allowed, possibly perturb diagonal entries)

**But** refinement generally required to recover accuracy

## Indefinite results: possible simple remedies

- Use relaxed pivoting (allow partial pivoting threshold parameter to reduce so that potentially less stable pivots are chosen).
- Static pivoting (no delays allowed, possibly perturb diagonal entries)

**But** refinement generally required to recover accuracy

**Problem:** solve is a bottleneck on multicore machines

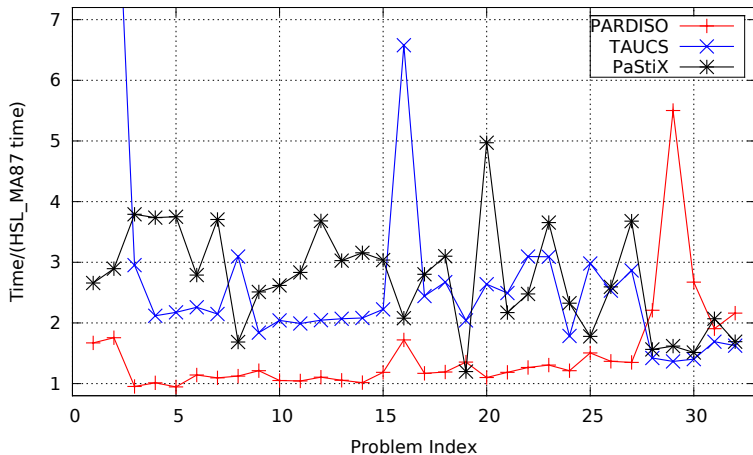
- Solve performs relatively small proportion of total flops
- But responsible for much higher proportion of memory traffic
- Memory bandwidth is limiting factor

Thus refinement is not cheap on multicore architecture.

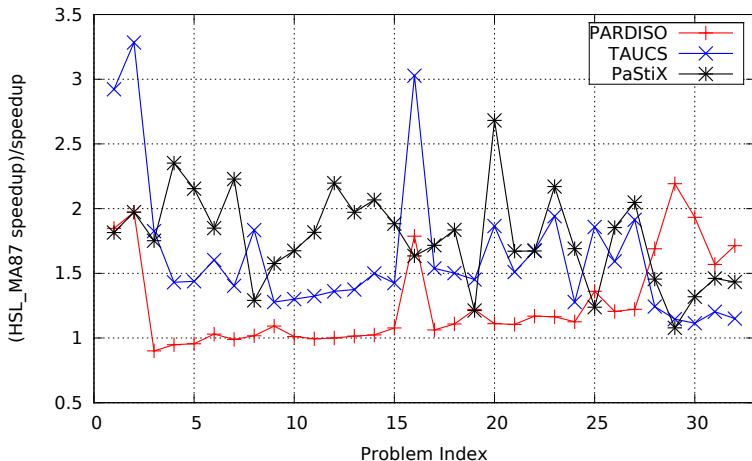
## Concluding remarks

- Extended DAG approach from dense positive-definite systems to sparse systems
- Very good results for factorizing positive-definite matrices on our 8-core machine
- Also good results for large indefinite problems provided there are few delayed pivots
- For some tough indefinite problems, further work needed to improve performance while maintaining stability.

# Positive-definite case: comparison with other solvers



# Positive-definite case: speedup ratios



## Indefinite case: comparison with PARDISO

Wall-clock times for factorization phase on 8 cores.

Problem	PARDISO	New code
Schenk_IBMNA/c-56	0.055	0.152
Boeing/crystk03	0.269	0.359
Cunningham/qa8fk	0.704	0.882
Schenk_AFE/af_shell10	13.2	11.7
Oberwolfach/bone010	174	88.3
GSH_indef/sparsine	159	44.4
PARSEC/Ga41As41H72	3020	1141