Direct solution of large sparse sets of equations on a multicore computer

John Reid, Jonathan Hogg, and Jennifer Scott

Rutherford Appleton Laboratory

Sparse Matrices for Scientific Computation, Cosener's House, Abingdon, 15 – 16 July, 2009.

Introduction

Multicore machines have become very common.

The NA Group at RAL has purchased an 8-core machine.

Its theoretical peak is 10/80 Glops on 1/8 cores.

Its `DGEMM` peak is 9.3/72.8, using MPI to run independent matrix multiplies.

We have been exploring how to make such a machine work well when factorizing sparse matrices. We use OpenMP.

The full symmetric positive-definite case

Jonathan Hogg has developed a Cholesky code, `HSL_MP54`, inspired by the work of Buttari, Langou, Kurzak, and Dongarra (2007).

The matrix is blocked and the computation is divided into tasks:

(i) **Factor**: $L_{kk}^T L_{kk} = A_{kk}$
(ii) **Solve**: $L_{ik} = A_{ik} L_{kk}^{-1}$
(iii) **Update**: $A_{ij} = A_{ij} - L_{ik} L_{jk}^T$

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}.

This leaves a lot of scope for tasks to be performed in parallel.

Priorities

The dependencies can be represented by a DAG (directed acyclic graph), with a node for each task.

Hogg found that complex priority schemes based on critical paths in the DAG offered very little benefit over the simple priority

(i) **Factor**
(ii) **Solve**
(iii) **Update**

Counts

For each block of L, count the tasks to be performed on it.

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

When count reaches 0 for block on the diagonal, spawn factorize task for it.

When factorize done, decrement count for each sub-diagonal block in its block column.

When count reaches 0 for off-diagonal block, spawn solve task.

When solve done, spawn updates made possible.

When update done, decrement count.

Performance in Gflops

threads	1	2	4	8
n				
500	5.6	8.6	13.4	17.7
2500	7.6	14.5	26.9	43.5
10000	8.6	17.1	33.6	61.9
Peak	9.3			72.8

The sparse case

Assume that a good pivot order is available.

There is freedom to alter this without changing the result, apart from round-off.

If l_{ij} is the first subdiagonal entry in column j, i must follow j in the pivot sequence (to ensure that a_{ii} is up to date).

Construct a tree with all such ij pairs as parent-child edges.

Each parent must wait for all its children.

Merging a node with its parent has little (or no) effect on fill-in.

Do this recursively (within reason) to give block eliminations at each node.

Result is the **assembly tree**.

Conventional parallelism

Usually rely on two levels of parallelism:

Tree-level parallelism: Independent subtrees processed in parallel.

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

Our idea: treat the whole computation as a set of factor, solve, and update tasks.

Data structure

Nodal matrix: columns of L that correspond to variables eliminated there, packed as a trapezoidal matrix.

Like holding the fully-summed parts of all the frontal matrices of the multifrontal method.

Divide this into blocks, wasting the upper triangular part of the blocks on the diagonal.

Hold each block by rows. Hold the blocks by block columns. Example:

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

Tasks

Factor and solve tasks are as in the full case.

Update of a block within the nodal matrix is as in the full case.

Update of a block not within the nodal matrix is different.

Updates between nodes

Updates not within the nodal matrix must be to an ancestor nodal matrix.

Consider the update

$$A_{ij} = A_{ij} - L_{ik} L_{jk}^{T}$$

to a block of such an ancestor nodal matrix.

L_{ik} consists of a set of contiguous rows of the nodal matrix. In general, they will correspond to a subset of the rows of A_{ij}.

Similarly, L_{jk} consist of a set of contiguous rows that correspond to a subset of the columns of A_{ij}.

The format (blocks held by rows) allows us to perform this as a single BLAS 3 to a buffer, followed by addition with indirect addressing.

Counts

As in the full case, keep track of the number of outstanding tasks for each block of *L*.

Initial values not so simple now and have to be calculating during analyse.

Effect of caching

As in the full case, keep track of the number of outstanding tasks for each block of L

When a task becomes ready, good idea to do it immediately with the same cache.

Hence we hold a small stack for each cache of tasks that are ready.

If it gets full, move bottom half to a global pool.

If it gets empty, get a task from global pool. If this, too, is empty, grab bottom half of biggest other stack.

Tasks in pool given priorities:

1. factorize
2. solve
3. update within node
4. update between nodes

Performance

We tested the code on 30 problems from Tim Davis' collection, with

orders from 20 to 1228 thousands,
max. front size from 1.9 to 10.7 thousands,
$nz(L)$ from 25 to 323 millions.

The speed-ups were:

2-core: close to 2
4-core: usually about 3.3, not less than about 3.
8-core: better than 6 on many large cases, not less than 5.

The actual 8-core speed was usually about 35 Gflops, less than 24 Gflops in only 2 cases.

Comparison with MUMPS and PARDISO

On one core

HSL_MA87 is broadly comparable with MUMPS and PARDISO.

On 8 cores

HSL_MA87 is broadly comparable with PARDISO, usually faster than MUMPS with median ratio about 2.

On one problem, the times were 17, 96, 120, resp.

Further work

We have begun to plan extension to the indefinite case, using 1×1 and 2×2 pivots subject to a threshold test.

This is based on our happy exprience with new indefinite kernel `HSL_MA64`.

Now have to wait for the whole of a block column to be ready before we can spawn updates.

A bit less scope for parallelism, but counts only needed for block columns instead of blocks.

Also need to allow for delayed pivots.

References

Buttari, A., Langou, J., Kurzak, J., and Dongarra, J. (2009). A class of parallel tiled linear algebra algorithms for multicore architectures. Parallel Computing **35**, 38-53.

Hogg, J. D. (2008) A DAG-based parallel Cholesky factorization for multicore systems. RAL-TR-2008-029.

Hogg, J. D., Reid, J.K., and Scott, J.A. (2008) A DAG-based sparse Cholesky solver for multicore architectures. RAL-TR-2009-004.