Communication Issues in Designing a Parallel Out-of-Core Multifrontal Linear Solver

Haim Avron and Anshul Gupta

IBM T. J. Watson Research Center

SIAM PP’12, February 2012
Out-of-Core Solvers

Solve $Ax = b$; $A$ is large, sparse and SPD; Shared memory platform
- Direct methods are robust and generally efficient
  ... but they do not scale well in terms of memory
- Problems are growing larger,
  and storage requirements grow more rapidly.
- Example: An $n^{1/3} \times n^{1/3} \times n^{1/3}$ grid is has $O(n)$ non-zeros
  ... but $O(n^{4/3})$ non-zeros in $L$ (optimal reordering)
- Some large problems may fit memory in sequential run
  ... but run out of memory when multithreaded.
- The out-of-core approach: keep some data on disk.
Why Parallel?

*This is SIAM Parallel Processing...*
Transferring data between disk and memory is communication between slow and fast memories, like in cache hierarchy.

However there are some important differences from cache hierarchy:

1. In cache hierarchy: very small fast memory (compared to peak memory use of the algorithms)
   In an OOC setting: fast memory is large, can even accommodate most of the memory use.

2. Bandwidth is more a concern than latency.

3. Problem: without special hardware, disk I/O is sequential by nature.
   - If I/O time is large we cannot hope for high speedups in factorization.
Background: Multifrontal Cholesky (sketch)

Input: Matrix $A$
Output: $L$ broken into **factor blocks**

1. **Symbolic factorization:**
   - form supernodes and etree
2. recursive-factorize(etree-root)

Recursive-factorize(etree-node)

1. For each child $k$ of etree-node call recursive-factorize($k$) and get **contribution block**
2. Form **frontal matrix** using input matrix and contribution blocks
3. Discard used contribution blocks.
4. Factor supernode to create **factor block**
5. Using front and factor block, create **contribution block**
Issues

- Communication complexity of the multifrontal method
- Communication costs and smart use of main memory
- Parallelism / Memory-usage / Communication trade-off
- Accuracy / Communication trade-off
Communication complexity of the multifrontal method
We want to use a multifrontal method since it allows the use of large blocks when calling BLAS.

However, Rotkin and Toledo (2004) observe that in an OOC setting the multifrontal method has a serious defect: there is a poor computation-to-communication ratio for tall-and-skinny supernodes.
Is it really a problem?

- Poor computation-to-communication ratio is also a problem for other memory hierarchies; it just another way to say that multifrontal is not communication optimal for some matrices.

But what about “real” matrices?

- Grigori, David, Demmel and Peyronnet 2010: On model problem (2D and 3D grids) the multifrontal method is communication optimal when supernodes are + separators (and some additional assumptions).
Communication costs and smart use of main memory
The communication complexity analysis is important, but not sufficient here.

- "Fast" memory is very large, analysis assumes it is small.
- The sequential nature of disk I/O implies that even constants matter.
- It applies only to the factor phase, but for the solve phase the situation is worse: it is completely I/O bound!

We want to make good use of main memory to reduce communication costs.
Minimizing I/O: Simple scheme

Assumptions:
- Factors blocks are always written to disk.
- A contribution block can be split between memory and disk.
- The active frontal matrix is held in memory.

Agullo (2008):
For a given postorder of the etree to get min I/O, write the least recently used contribution block data when some I/O is necessary.
Why factors on disk?

- Writing a word from a factor block to disk will cause a one word of W+R.
- Keeping a factor block word in memory and writing a contribution word byte instead will cause:
  - at least one W+R for that word
  - possibly many W+R of bytes from other contribution blocks
- It is better to evict a words from factor blocks first.
- So: if maximum stack size bigger than memory, factor blocks should be kept on disk.
Why not factors on disk?

However,

- Analysis no longer correct when writing/reading complete (factor/contribution) blocks
- Maximum stack size may be reached midway
  - Space to store factors after max was achieved?
- In parallel, only an upper bound of maximum stack size
- Not the same cost of I/O at different phases:
  - Factor phase is computationally bound...
    I/O can be hidden using computation
  - Solve phase is I/O bound...
    I/O cannot be hidden

We try to balance: keep many factor blocks in memory, but not to send too many contribution blocks to disk
Why not factors on disk?

However,

- Analysis no longer correct when writing/reading complete (factor/contribution) blocks
- Maximum stack size may be reached midway
  - Space to store factors after max was achieved?
- In parallel, only an *upper bound* of maximum stack size
- Not the same cost of I/O at different phases:
  - Factor phase is *computationally bound*...
    - I/O can be hidden using computation
  - Solve phase is *I/O bound*...
    - I/O cannot be hidden

We try to balance: keep many factor blocks in memory, but not to send too many contribution blocks to disk
Our memory management scheme

Large memory buffer – allocated at the beginning

Task private memory

Expanding space for factors:
- Filled from right to left
- Block location determined statically.
- Preference for top of etree.

Shared buffer for contribution block stacks. “L2” buffer; the “L1” thinks it is on disk.

“Top” of contribution block stack. “L1” buffer.

Working space:
Current frontal and largest sibling contribution block.

Assumptions:
1) Tasks are continuous subtrees
2) Blocks are processed as a unit
3) Frontal matrices are always processed in-core
4) Extend-add always done in-core
Parallelism / Memory-usage / Communication tradeoff
Minimum memory

Assumptions:
- All frontal matrices are processed in memory
- Contribution blocks loaded as one unit

We need at least enough in-core memory to hold any single frontal matrix and largest child contribution.

In parallel: several such pairs can exist together

- Increase in #threads increases minimum memory
- ... which decreases space for buffering factor/contribution blocks
- ... which increases I/O and therefore time for I/O
- ... which reduces parallelism
Assumptions:

- All frontal matrices are processed in memory
- Contribution blocks loaded as one unit

We need at least enough in-core memory to hold any single frontal matrix and largest child contribution.

In parallel: several such pairs can exist together

Increase in #threads increases minimum memory
... which decreases space for buffering factor/contribution blocks
... which increases I/O and therefore time for I/O
... which reduces parallelism
Countermeasures

1. Avoid too aggressive parallelism, i.e. spawning many tasks and relying on work-stealing.

2. “Pushing down” parallelism to
   - fit into available in-core memory
   - reduce memory requirements and allow more factor blocks to be kept in memory (still experimenting with that)
Accuracy / Communication Trade-off
Reducing Accuracy for Reducing Communication

- Full double precision needs 8 bytes, single precision 4 bytes.
- Using single precision reduces I/O by more than 50%.
  - Data written to disk is overflow, which is reduced.
- When I/O is big, saving can be substantial, especially in a parallel setting.
- If I/O is sufficiently reduced, accuracy can be recovered using iterative refinement.

Instead of just reducing accuracy we use (lossy) compression.
Compression

Underlying observations:
- Many zeros early in the factorization, very few later.
- Values tend to have a small exponent range, and few unique exponents.

Compression method:
- Compression is done on both contribution blocks and factor blocks, but they may be amalgamated or broken.
- In each block, analyze the exponents and compress them (e.g., fixed length encoding).
- Round additional least significant bits to meet compression goal.
Preliminary Experimental Results
Speedup

(no compression)

Speedup of OOC solver

2 Cores
4 Cores
8 Cores

Matrix (sorted by 1 CPU time)
WSMP OOC vs. IC - sequential and 8 cores

(no compression)
Effect of compression

(8 cores, a large matrix)
Comparision to other libraries

(no compression)
Parallelizing an out-of-core solver is challenging because communication is not only slow but also sequential.

Decent speed-up are achievable by smart use of memory: not all communication affect scalability the same, so it better to keep factors in memory.

Main contributions:

- A novel method to manage memory in a parallel out-of-core solver.
- "Pushing down of parallelism" to enable utilization of very large processor counts without increasing in-core memory.
- Method to determine which blocks are kept in memory and which on disk.
- The use of compression to speed up the solve phase.

OOC will be available in the next release of WSMP. Libraries downloadable from:
Thank You! Questions?