Parallel out-of-core extension to ScaLAPACK

Ed D’Azevedo (e6d@ornl.gov)
Computer Science and Mathematics Division
Oak Ridge National Laboratory
Oak Ridge, TN 37831

Old Dominion University
April 15, 2005
Acknowledgements

- Jack Dongarra, Director of Innovative Computing Laboratory (ICL) at the University of Tennessee, also Distinguished Research Staff at ORNL for discussion and guidance.
- Piotr Luszczek, in ICL/UTK, for implementation of check-point and restart capability.
- DARPA, DOE SciDAC (Scientific Discovery through Advanced Computing) and ORNL Laboratory Directed Research and Development (LDRD) Program.
- This research was performed at the Oak Ridge National Laboratory, managed by UT-Battelle, LLC, for the U.S. Department of Energy under contract DE-AC05-00OR22725.
Overview

- Background on Out-of-core computation
- Background on ScaLAPACK
- Design on Software
- Design Decision on I/O
- Design on Algorithm
- Check-point and Restart
- Performance
Motivation for Out-of-core Computation

- Solve very large dense matrix problems several times larger than available memory.
- Disks storage is much cheaper than memory, e.g. 2 GBytes PC2700 memory cost $200, 200 GBytes SATA drive cost $100 (factor 200).
- Scheduling of out-of-core computation and I/O should be much more efficient than just “paging”. Large (many MBytes request) I/O requests versus small (4 KBytes) pagesize.
Rough Estimate for LU Factorization

- Typical x86 Linux box with 3.2 GHz Pentium 4 and 2 GBytes of memory. The processor can achieve about 5 Gflops/s with SSE2 optimized BLAS. (64-bit Opteron multi-processor workstation may accommodate up to 16 GBytes.)

- Largest dense complex*16 matrix is about \( N = 11585 \). LU factorization requires about \( O(2.67N^3) \) flops, or about 14 min at 5 Gflops/s.

- If I/O system can sustain 40 MBytes/s transfer, then reading/writing 2 GBytes takes about a minute.
Rough Estimate for Larger problems

- Let $M$ denote the amount of memory, then problem size $N = O(\sqrt{M})$. Work increases as $O(N^3)$ or $O(M^{3/2})$.
- If we increase available storage by 4 times, the work (or runtime) increases by $4^{3/2} = 8$. Similarly, 9 times the storage requires about $9^{3/2} = 27$ times the work.
- From previous example, 9 times storage is only 18 GBytes and takes very roughly about $27 \times 14$ min or 6.3 hr (378 min) (assuming I/O overhead is not significant).
- Note it requires roughly 9 min to read or write 18 GBytes.
Another approach is to solve a large problem in parallel. Suppose we have a 32 processor cluster and each processor has its own local disk, then largest in-core complex*16 dense matrix is $N = 92682$, and takes about 3.7 hrs for LU factorization.

Similarly, 9 times the storage would require about $27 \times 3.7 \text{ hr} \approx 4.2 \text{ days}$.

Adding more processors also add more memory, more disks and higher I/O bandwidth. Some distributed memory machine, say IBM SP, has a dedicated parallel file system. However, I/O bandwidth is fixed and may not scale with more processors.
Out-of-core Computation


- Thomas algorithm (LU factorization) for block tridiagonal system where the “U” factors are stored on disk.

- Large dense matrices arise from many applications:
  - Electromagnetic scattering off aircraft
  - Response of plasma to radio frequency in fusion
  - Boundary Element method
  - Least-squares calculations in Geodesic Application

- Fast Multipole Method may be more appropriate in some cases.
• Parallel library for dense matrix problems. Subroutine name, calling sequence and arguments modeled after LAPACK. Written mostly in Fortran 77 and some C for portability.


• Relies on Parallel Basic Linear Algebra Subroutines (PBLAS) for performance and Basic Linear Algebra Subroutines (BLACS) for communication.

• Variant of LU factorization used as benchmark in TOP 500 list (www.top500.org) of fastest computers.

• PLAPACK (Parallel Linear Algebra Package) developed at University of Texas at Austin also offers similar capability but uses a objected oriented approach.
Matrix Descriptor

- Two-dimensional block cyclic distributed storage on a two-dimensional processor grid.
- Dense LU prefers a nearly square grid since $p \times 1$ grid has high communication cost in determining pivot row, but $1 \times p$ grid has serial bottleneck.
- Matrix block size, communication context and other information on distribution stored in matrix descriptor.
- Large block size MB may yield good serial performance but reduce opportunity for concurrent computation. Very small block size can incur high message overhead.
2-DIMENSIONAL BLOCK CYCLIC DISTRIBUTION

Global (left) and distributed (right) views of matrix
Utility Routines

DESCINIT  Setup matrix descriptor

NUMROC   Calculate locate extent or storage

INFOG2L  Calculate local indices and processor coordinates from
global matrix indices

INDXL2G  Calculate global matrix index from local index

INDXG2L  Calculate local matrix index from global index

INDXG2P  Calculate processor coordinate from global index.
PBLAS

- Modeled after serial BLAS so that parallel code looks similar to serial code.

- Original version 1 had alignment restrictions but version 2 (based on PhD work by Antoine Petitet, LAPACK Working Notes 128 at www.netlib.org) has removed all alignment restrictions.

- PBLAS version 2 has internal algorithmic blocking and multiple optimized algorithms for special cases.

- For example, PZTRSM triangular solve checks whether it is more efficient to transfer the RHS vectors or transfer a small triangular matrix.

- PZGEADD can be used to copy submatrices. This is usually more efficient than multiple calls to PZCOPY.
**Out-of-core Software**

- Calling sequence and naming convention modeled after ScaLAPACK.
- Generalized PBLAS like layer to operate on out-of-core matrix.
- Idea: Read large submatrices into memory, use in-core ScaLAPACK routines. Perform sufficient work in memory so that I/O overhead is acceptable.
- Generalization of ScaLAPACK matrix descriptor with extra fields to store extra information about disk I/O.
Software

- User specify amount of in-core memory (ASIZE) available (without causing paging) for out-of-core software. Code automatically switches to in-core solver if ASIZE is sufficiently large to hold entire matrix.

- Implementation for LU, QR, Cholesky factorization and solver.

- Software available at http://www.netlib.org/scalapack/prototype/
Example

1  * LAPACK
2  CALL ZGETRF( M, N, A(IA,JA), LDA, IPIV, INFO )
3
4
5  * in-core ScaLAPACK
6  CALL DESCINIT(DESCA,M,N,MB,NB,RSRC,CSRC,ICONTEXT, &
7       LDA,INFO)
8  CALL PZGETRF( M, N, A, IA, JA, DESCA, IPIV, INFO )
9
10
11  * Out-of-core extension
12  CALL PFDESCINIT(DESCA,M,N,MB,NB,RSRC,CSRC,ICONTEXT, &
13       IODEV,’DISTRIBUTED’,MMB,NNB,ASIZE,FILENAME,INFO)
14  CALL PFZGETRF( M, N, A, IA, JA, DESCA, IPIV, INFO )
Design of I/O

- Need an efficient and portable I/O interface. Currently uses C `read`, `write`, `lseek`.

- Use multiple files to get around maximum 2 GBytes file limit (32-bit `lseek` file pointer)

- No asynchronous I/O:
  - No simple, portable implementation
  - Extra buffer space can be more efficiently used for computation
  - No help if application is already compute bound or I/O bound
Layout of Data on Disk

- `lseek` can be expensive, especially on a parallel file system. Fortran column oriented storage is not suitable for reading in wide submatrices. Similarly, image of entire distributed 2D block-cyclic Scalapack matrix is also not efficient for copying submatrices.

- Use record-oriented storage. Each record is an image of an in-core Scalapack matrix of size $\text{MMB}=L_1 \times MB \times P$ by $\text{NNB}=L_2 \times NB \times Q$ on $P$ by $Q$ processor grid.

- Each processor concurrently reads or writes $(L_1 \times L_2)$ blocks of size $(MB \times NB)$ to get good performance.
Three Modes for I/O

**Distributed:** Each processor has its own set of files with unique filenames on a local (maybe shared on SMP) file system.

**Interleaved:** All processors write to a single large file on a parallel file system. Data from each record is interleaved to facilitate caching by the parallel file system.

**Shared:** All processors write to a single large file on a parallel file system. Similar to concatenation of many distributed files to reduce the total number of file descriptors.
Parallel Read and Write

- Subroutines ZLAREAD and ZLAWRITE to transfer a $M \times N$ submatrix between disk and in-core ScaLAPACK matrix.

- Data record is first transferred (or copied) to an intermediate buffer, which is also a ScaLAPACK in-core matrix.

- Software perform message communication and memory copy if I/O is not aligned on same processors. Use PBLAS PZGECOPY for data transfer.

- Network communication is commonly faster than disk I/O.
Disk cache

- Some system perform very aggressive caching of disk in memory. For small problems, or using only a subset of processors on a shared memory machine, this may yield unrealistically high I/O bandwidth (basically memory to memory copy from cache).

- There is little reuse of data unless a very significant fraction of disk file can be cached. However, the memory for this buffer or cache memory can be more effectively used by out-of-core solver.

- There is no simple and portable way to discover how much in-core memory is actually available without paging.
Algorithm

- Algorithm keeps two column panels (label X, Y) in memory.
- If pivoting is not required, rectangular subblocks might be used for Cholesky factorization. This package uses column panels even for QR and Cholesky factorization.
- Panel X should be NNB columns wide to get good I/O performance and panel Y should take up the remaining storage to be as wide as possible.
• Panel X acts as a buffer to hold and apply previously computed factors to panel Y.

• Once all previous updates are applied, panel Y is factored using in-core ScaLAPACK and written out to disk.

• I/O volume (and time) is dominated by multiple reading for panel X. Number of passes depends on the width of panel Y.

• Minimum in-core memory is for 2 panels, each panel is a $N \times NNB$ in-core ScaLAPACK matrix.
LU Factorization

- Block partitioned matrix, suppose there is no pivoting (for simplicity)

\[
A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \approx \begin{pmatrix} L_{11} \\ L_{21} \\ L_{22} \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} \\ U_{12} & U_{22} \end{pmatrix}
\]

- Factor first panel

\[
A_{11} = L_{11}U_{11}, \quad L_{21} = U_{11} \setminus A_{21}
\]

- Update second panel from already computed factors \(L_{11}, U_{11}, L_{21},\)

\[
U_{12} = L_{11} \setminus A_{12}, \quad \tilde{A}_{22} = A_{22} - L_{21}U_{12}, \quad \tilde{A}_{22} = L_{22}U_{22}
\]
LU Factorization

- First panel, combine storage for X and Y
  1. LAREAD: read in part of original matrix
  2. PxGETRF: ScaLAPACK in-core factorization
     \[
     \begin{pmatrix}
     L_{11} \\
     L_{21}
     \end{pmatrix}
     (U_{11}) \leftarrow P_1
     \begin{pmatrix}
     A_{11} \\
     A_{21}
     \end{pmatrix}
     \]
  3. LAWRITE: write out factors
LU Factorization

- Let panel $Y$ hold $A_{12}$, and $A_{22}$, panel $X$ hold $L_{11}$, and $L_{21}$
  1. **LAREAD**: read in part of factor into panel $X$
  2. **LAPIV**: physically exchange rows in panel $Y$ to match permuted ordering in panel $X$

\[
\begin{pmatrix}
\tilde{A}_{12} \\
\tilde{A}_{22}
\end{pmatrix}
\leftarrow
P_1
\begin{pmatrix}
A_{12} \\
A_{22}
\end{pmatrix}
\]

3. **PxTRSM**: triangular solve to compute upper triangular factor

\[
U_{12} \leftarrow L_{11}^{-1} \tilde{A}_{12}
\]

4. **PxGEMM**: update remaining lower part of panel $Y$

\[
\tilde{A}_{22} \leftarrow \tilde{A}_{22} - L_{21} U_{12}.
\]
LU Factorization

- After all updates are applied to $\tilde{A}_{22}$, we apply $\text{PxGETRF}$ to compute LU factors in panel $Y$

  $$L_{22}U_{22} \leftarrow P_2\tilde{A}_{22}$$

- $\text{LWRITE}$ to write panel $Y$ to disk

- A final extra pass over the computed lower triangular $L$ matrix may be required to rearrange the factors in the final permutation order

  $$\tilde{L}_{12} \leftarrow P_2L_{12}$$
Left-looking Algorithm

- All panel X are to the “left” of panel Y so that each panel Y is read in once and written out once.

- A “right-looking” algorithm immediately updates all panels to the right of panel X once X is factored. This would require higher I/O volume to repeatedly read in and write out panel Y.

- In-core ScaLAPACK uses a right-looking algorithm to expose more work for parallelism.
Check-point and Restart

- Runs may take several days that may exceed time limit in batch queue policy, or approach MTBF on linux cluster built with off-the-shelf components.
- Recovery or restarting is conceptually simple since the factors and partial results are still on disk.
- May require redundant (or duplicate) computation from a consistent check-point.
- Two types of termination: (i) expected termination (time limit in batch queue) (ii) unexpected termination (system crash due to software or memory error).
**Approach**

- Simulate out-of-core computation to generate “micro-instructions" in file. The instructions are easily mapped to high-level subroutine calls such as read in panel (**ZLAREAD**), write out panel (**ZLWRITE**), perform update (**PBLAS**), in-core factorization (**PZGETRF**).

- Simple driver to process (or play back) list of instructions. Driver can write out partial results and last instruction before stopping.

- Details: driver look ahead in instruction list to find next write instruction in check-point; driver look back to find read command to restore panel in memory.
Micro-instructions

zlaread__
1 0 16000 8316 99 99 0 0 8019 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0
0 0 11 16000 8316 1 1 1 1 1 0 0 0 0 0 0 0 (0.,0.) (0.,0.)
pzgetrf__
1 0 16000 8316 99 99 0 0 8019 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0
0 0 16000 8316 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 (0.,0.) (0.,0.)
zlawrite__
1 0 16000 8316 99 99 0 0 8019 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0
0 0 11 16000 8316 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 (0.,0.) (0.,0.)
zlaread__
Writing out in 2-Steps

- The factored panel is first written out to a temporary location. Out-of-core matrix is not affected if this fails.

- If this is successful, the out-of-core matrix is then updated. Even if a machine crash corrupts the matrix, the intact data in the temporary location can be used for recovery.

- Details: other (empty) temporary files are used to mark progress, e.g. 00143a.dat (00143b.dat) indicate first (second) write is successful for instruction 143.
Limitations

- Need to restart with same set of processors if a local file system (e.g. /tmp on linux cluster) is used.

- Assume all data is automatically “synced” or flushed out to disk after the file is closed. Delayed writes may hinder recovery on unexpected machine crash since the data may still be in volatile memory and not written to disk.
Performance Results

- Runs performed on a Linux cluster where each node is a dual AMD Opteron 242 (1.6 GHz) processor with 2 GBytes of memory. Each node equipped with Maxtor EIDE-133 120 GByte 7200 RPM disk with 8 MByte internal cache and connected with a fast quadrics switch.

- Code compiled with g77 but linked to optimized BLAS by Kazushige Goto (http://www.cs.utexas.edu/~flame/goto/). The optimized BLAS library achieves about 2.7 Gflops/s in $1000 \times 1000$ matrix multiply in ZGEMM and about 2.2 Gflops/s in LAPACK ZGETRF.

- In-core ScaLAPACK PZGETRF for $N=2200$ (MB=NB=50) on 4 cpus achieved about 2.15 Gflops/s per processor. Two MPI tasks are spawned on each node to take advantage of dual processors.
Run with 16 cpus

- N=60000, 512 MBytes for ASIZE, MB=NB=50, L1=L2=4, P=Q=4. Overall performance about 2.0 Gflops/s per cpu. Total matrix size is about 54 GBytes or about 3.4 GBytes/cpu.

<table>
<thead>
<tr>
<th>Routine</th>
<th>No. of Calls</th>
<th>Time (percentage)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZEXPFGETRF</td>
<td>1</td>
<td>17775s (100%)</td>
</tr>
<tr>
<td>ZLAREAD</td>
<td>922</td>
<td>1266s (7.1%)</td>
</tr>
<tr>
<td>PZGETRF</td>
<td>7</td>
<td>2005s (11.3%)</td>
</tr>
<tr>
<td>ZLAWRITE</td>
<td>13</td>
<td>465s (2.6%)</td>
</tr>
<tr>
<td>PZLAPIV</td>
<td>27</td>
<td>243s (1.4%)</td>
</tr>
<tr>
<td>PZTRSM</td>
<td>3636</td>
<td>15s (0.1%)</td>
</tr>
<tr>
<td>PZGEMM</td>
<td>3636</td>
<td>12877s (72.4%)</td>
</tr>
</tbody>
</table>
Extra Write for Recovery

- Overall runtime increased to about 19128s if an extra write is performed for recovery from unexpected termination.
- Performance is about 1.88 Gflops/s per cpu.
- Increase of about 7.6% (from 17775s).
- Single call to `PZGETRF` (without micro-instructions) took about 18368s. Performance at about 1.96 Gflops/s per cpu.
Performance on Hydra

- 8 1200 MHz UltraSPARC IV processors with 64 GBytes of RAM
- ZGEMM on $1000 \times 1000$ achieved about 2.13 Gflops/s, ZGETRF achieved about 1.67 Gflops/s.
- In-core ScaLAPACK PZGETRF on $N=8000$, $MB=NB=99$, $2 \times 2$ processor grid achieved about 1.51 Gflops/s per cpu.
- Write bandwidth about 11 MBytes/s per cpu.
- Largest complex*16 problem is $N=63000$. If each cpu can sustain 1.5 Gflops/s in ScaLAPACK PZGETRF, the factorization will take about 15.5 hr.
**2 × 2 Processors on Hydra**

- N=30000, 512 MBytes for ASIZE, MB=NB=50, L1=L2=2.
  Overall performance about 1.37 Gflops/s per cpu. Total matrix size is about 13.7 GBytes or about 3.4 GBytes per cpu.

<table>
<thead>
<tr>
<th>Routine</th>
<th>No. of Calls</th>
<th>Time (percentage)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZEXPFGETRF</td>
<td>1</td>
<td>13203s (100%)</td>
</tr>
<tr>
<td>ZLAREAD</td>
<td>610</td>
<td>562s (4.3%)</td>
</tr>
<tr>
<td>PZGETRF</td>
<td>8</td>
<td>1427s (10.8%)</td>
</tr>
<tr>
<td>ZLAWRITE</td>
<td>15</td>
<td><strong>1277s</strong> (9.7%)</td>
</tr>
<tr>
<td>PZLAPIV</td>
<td>35</td>
<td>303s (2.3%)</td>
</tr>
<tr>
<td>PZTRSM</td>
<td>2380</td>
<td>23s (0.2%)</td>
</tr>
<tr>
<td>PZGEMM</td>
<td>2380</td>
<td>9609s (72.8%)</td>
</tr>
</tbody>
</table>
Summary

- Out-of-core software enable larger problems to be solved.
- The out-of-core software has similar interface as ScaLAPACK and LAPACK.
- The I/O overhead accounts for only a small fraction of overall time and yields good performance.
- Check-point and restart capability convenient for long jobs.