Lessons learned from the development of a parallel sparse direct solver

Kyungjoo Kim

Aerospace Engineering and Engineering Mechanics
The University of Texas at Austin

Sep 6, 2013
Outline

1. Source of sparse matrix: *hp*-FEM

2. Parallelization strategy

3. Heterogeneous architectures: multi-level matrix blocking

4. Discussion

5. Summary
Source of sparse matrix: *hp*-Finite Element Method (FEM)

**Figure**: A sparse system of equations is generated based on a FE-mesh.\(^1\)

\(p = 1, \text{ vertex} \quad p = 2, \text{ edge} \quad p = 3, \text{ edge} \quad p = 3, \text{ face}\)

**Figure**: Example of high order basis functions.

\(^1\)The airfoil mesh is obtained from Matlab.
**hp-FEM delivers fast convergence rate**

Example: projection of the manufactured solution: \( \psi = \sin(x) \cos(y)z \)

\[ p=4, \ # \ of \ FEs=1, \ err=1.27\% \quad p=1, \ # \ of \ FEs=32,768, \ err=1.19\% \]

**Figure**: For a smooth solution, the use of high \( p \) delivers a fast convergence rate.
Source of sparse matrix: \textit{hp}-FEM

Application: wave propagation problems

\textbf{Figure :} \textit{Underwater acoustics with a rough seabed} \textsuperscript{2}, approximated by \( p=6 \), \# of elements= 1,130 (368 in the domain of interest), \# of DOFs= 200k.

\textsuperscript{2}The image is produced by Jeffrey Zitelli.
Application: wave propagation problems

Figure: Underwater acoustics with a rough seabed, approximated by $p=6$, # of elements = 1,130 (368 in the domain of interest), # of DOFs = 200k.

\footnote{The image is produced by Jeffrey Zitelli.}
Sparse system of equations generated by *hp*-FEM

- All operations are essentially dense.

<table>
<thead>
<tr>
<th>Node Type</th>
<th>Edge</th>
<th>Face</th>
<th>Interior</th>
</tr>
</thead>
<tbody>
<tr>
<td># of DOFs</td>
<td>$O(p)$</td>
<td>$O(p^2)$</td>
<td>$O(p^3)$</td>
</tr>
</tbody>
</table>

(a) High order FE

(b) Element matrix

**Figure:** *The shape of an unassembled element matrix.*
Sparse system of equations generated by $hp$-FEM

- All operations are essentially dense.

<table>
<thead>
<tr>
<th>Node Type</th>
<th>Edge</th>
<th>Face</th>
<th>Interior</th>
</tr>
</thead>
<tbody>
<tr>
<td># of DOFs</td>
<td>$O(p)$</td>
<td>$O(p^2)$</td>
<td>$O(p^3)$</td>
</tr>
</tbody>
</table>

(a) $p = 1$, $nz = 226,981$

Figure: Nonzero patterns keeping the same system DOFs.

(b) $p = 4$, $nz = 1,771,561$
Multifrontal factorization in FEM

- Characterized by **recursive** procedure on the assembly tree.
- Performs **supernodal** elimination and assembly for each frontal matrix.
- Converts the sparse matrix factorization into **multiple dense subproblems**.

**Figure**: The factorization is completed ascending the assembly tree.
Two-level parallelism

- High degree **tree-level** parallelism on leaves.
- Increasing opportunity in **matrix-level** parallelism.

Asynchronous task execution in harmony with two-level parallelism

- Load imbalance due to irregular task sizes.
- Bandwidth-bounded tasks on leaves vs compute-bounded tasks nearby the root.
Two-level parallelism

- High degree **tree-level** parallelism on leaves.
- Increasing opportunity in **matrix-level** parallelism.

Asynchronous task execution in harmony with two-level parallelism

- Load imbalance due to irregular task sizes.
- Bandwidth-bounded tasks on leaves **vs** compute-bounded tasks nearby the root.
Fine-grained task generation: algorithms-by-blocks

E. Chan et al., 2007., Satisfying your dependencies with Supermatrix.

G. Quintana-Ortí et al., 2009., Programming matrix algorithms-by-blocks for thread-level Parallelism.
Hierarchical DAG scheduling

Tasks are **locally** analyzed and **globally** ordered.

- Tree-level tasks (priori known structure) are generated via parallel post-order tree traversal.
- Fine-grained tasks are generated by using algorithms-by-blocks.
- Tasks are hierarchically ordered together with multiple Directed Acyclic Graph (DAG) schedulers.
Figure: An example of hierarchical DAGs.
Strong scale

*Figure*: Factorization phase for fixed $p = 4$ with a reference time of the sequential UHM solver.
Scheduling tasks to multiple GPUs
We want to achieve portable performance with manageable programming complexity.

Challenges:

- A large front may not fit into a **small device memory (6 GB)**.
  → A large matrix is decomposed of blocks; only computing blocks are transferred to devices.

- **Different programming models** can be used.
  → Blocks are computed via vendor provided libraries (e.g., MKL, CUBLAS).

- Efficient workload balancing among **asymmetric computing units**.
  → Workloads are dynamically partitioned based on device performance.
Scheduling tasks to multiple GPUs

We want to achieve portable performance with manageable programming complexity.

Challenges:

- A large front may not fit into a small device memory (6 GB).
  → A large matrix is decomposed of blocks; only computing blocks are transferred to devices.
- Different programming models can be used.
  → Blocks are computed via vendor provided libraries (e.g., MKL, CUBLAS).
- Efficient workload balancing among asymmetric computing units.
  → Workloads are dynamically partitioned based on device performance.

![Figure: Multi-level matrix blocking improves unit performance and efficiency.](image-url)
Task handling: bulk-synchronous approach

Suppose that the target architecture has four computing units and one GPU whereas their performance ratio is 1:3.

- Dense subproblems are computed within a sequence of supersteps.
- Each superstep consists of tasks that can be executed concurrently.
- Tasks are dispatched to heterogeneous computing units in a round-robin fashion (easy to exploit multiple GPUs).
Dense problems: two Fermi GPUs

Figure: Dense LU factorization without pivoting accelerated by multiple GPUs.
Sparse factorization: two Fermi GPUs

<table>
<thead>
<tr>
<th>Cores</th>
<th>GPUs</th>
<th>Time [sec]</th>
<th>GFLOP/sec</th>
<th>Speed-up vs 1 core</th>
<th>Speed-up vs 12 cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>291</td>
<td>11.13</td>
<td>1.00</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>213</td>
<td>15.21</td>
<td>1.36</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>85</td>
<td>38.11</td>
<td>3.42</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>45</td>
<td>71.98</td>
<td>6.46</td>
<td>-</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>33</td>
<td>98.16</td>
<td>8.81</td>
<td>1.00</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>23</td>
<td>140.84</td>
<td>12.65</td>
<td>1.43</td>
</tr>
<tr>
<td>12</td>
<td>2</td>
<td>19</td>
<td>170.50</td>
<td>15.31</td>
<td>1.69</td>
</tr>
</tbody>
</table>

Table: Sparse LU with partial pivoting accelerated by multiple GPUs.
Discussion: runtime parallelism vs structured parallelism

How can we put runtime parallelism in harmony with structured parallelism?

**Runtime task parallelism**

**Structured parallelism**

<table>
<thead>
<tr>
<th></th>
<th>Runtime</th>
<th>Structured</th>
</tr>
</thead>
<tbody>
<tr>
<td>Granularity</td>
<td>Fine</td>
<td>Finer</td>
</tr>
<tr>
<td>Concurrency</td>
<td>Out-of-order scheduling</td>
<td>Dependent on algorithms</td>
</tr>
<tr>
<td>Locality</td>
<td>Data affinity scheduling</td>
<td>Predefined data partitions</td>
</tr>
<tr>
<td>Parallel overhead</td>
<td>High</td>
<td>Low</td>
</tr>
</tbody>
</table>
Discussion: runtime parallelism vs structured parallelism

How can we put runtime parallelism in harmony with structured parallelism?

Requirements in DLA interface:

- DLA algorithms are designed with abstract communicators.
- Tasks are generated from DLA algorithms with light-weight communicators.
- Runtime resource manager dynamically controls resource allocation for given tasks.
Lessons learned

Increased reliance on DLA libraries.

- Application problem is characterized by **dense** block sparse matrix.
- Supernodal sparse factorization forms a tree of **dense** problems.

High performance computing in the application context.

- Multi-level tasking effectively combines multifrontal factorization with runtime task parallelism.
  ✓ high performance of DLA libraries → high performance sparse direct solver.
- Dynamic task subdivision approach provide reduce the number of data transfer to devices and provide a suitable granularity to devices.

Can BLIS provide building blocks for users to build their own parallelism?

Thank you.