Sparse Triangular Solve in UPC

By Christian Bell and Rajesh Nishtala
Motivation

• A common but irregular mathematical operation occurring in linear algebra is the Sparse Triangular Solve (SpTS).
  – Solve for $x$ in $Tx = b$ where $T$ is a lower triangular sparse
  – Used after sparse Cholesky or LU factorization to solve sparse linear systems

• Irregularity arises from dependence

• Hard to parallelize
  – dependence structures only known at runtime
  – must effectively build dependence tree in parallel
Algorithm Description

• To solve for \( x \) in \( T x = b \) (\( T \) is lower triangular)
  
  \[
  \begin{align*}
  &\text{for } r=1:n \{ \\
  &\quad x(r) = b(r); \\
  &\quad \text{for } c=1:r \\
  &\quad\quad x(r) = x(r) - T(r,c) \cdot x(c); \\
  &\quad \}
  \end{align*}
  \]

• Key takeaways
  
  – To solve \( x_r \) you depend on all values of \( x \) before it
  
  – rows can be partially solved by knowing which values of \( x_c \) are valid
Dependency Graph

Matrix

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4</td>
</tr>
</tbody>
</table>

Dependence Graph

0 → 4 → 1 → 3 → 2 
7 → 5 → 6
Data Structure Design

• Allow more startup time analysis of matrix so that the solve is faster
• Build the dependence graph in parallel
• Support $O(1)$ lookup during solve time
• $O(1)$ operations made easy by UPC
Solve Methodology

- **Producer / Consumer Relationship**
  - “consume” x vector in $Tx = b$ to produce a new $x_j$ variable.
  - “production” causes generation of signal to every processor waiting on $x_j$

- **Difficult with two-sided model of MPI**
  - allows you to effectively leverage one-sided communication available in UPC

- **Avoid synchronization**
  - by knowing a priori what part of other threads address space you can safely write into.
  - very difficult to get right through MPI
Performance (1)

bmw matrix m=141347 n=141347 nz=5066530
Pentium III Xeon / Myrinet

Number of Threads vs. Speedup
- Blocking
- Non Blocking

Graph showing the speedup for different numbers of threads: 1, 2, 4, 8, 16, and 32 threads. The graph compares blocking and non-blocking approaches, with non-blocking showing a higher speedup for higher numbers of threads.
Performance (2)

Speedups Across Matrices (Pentium III 866MHz/ Myrinet cluster)

<table>
<thead>
<tr>
<th>Matrix</th>
<th>Number of Nonzeros</th>
<th>Number of Matrix Entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>BMW</td>
<td>~141k</td>
<td>~5M</td>
</tr>
<tr>
<td>LHR</td>
<td>~70k</td>
<td>~3.1M</td>
</tr>
<tr>
<td>MEMPLUS</td>
<td>~18k</td>
<td>~0.33M</td>
</tr>
</tbody>
</table>
Conclusions and Future Work

• A new style of programming for an old problem
• Leverage one-sided messaging not easily available in MPI
• Integrate into libraries such as SuperLU