Parallelizing Gauss-Seidel Using Full Sparse Tiling

Michelle Mills Strout

CSCAPES Seminar
May 15, 2007
Goal: Make computations over meshes fast

- Iterative smoothers such as Gauss-Seidel dominate the execution time of finite element applications
- Need to effectively utilize the memory hierarchy
- Performance should scale to multiple processors
Full Sparse Tiling to the Rescue!

• Problem
  - Compile-time data reordering and computation scheduling to improve data locality and parallelism is not possible due to irregular memory references \(A[B[i]]\)

• Solution
  - Inspector/executor strategies perform data and computation reordering at runtime
  - Full sparse tiling is one such strategy that improves performance by exploiting parallelism, intra-iteration data reuse, and inter-iteration data reuse
Gauss-Seidel Iteratively Solves $Au = f$

- $u$ is a vector of unknowns
- $A$ is a sparse matrix stored in the compressed sparse row format (CSR)

Matrix Graph

```
Matrix Graph

Gauss-Seidel Iteratively Solves $Au = f$

- $u$ is a vector of unknowns
- $A$ is a sparse matrix stored in the compressed sparse row format (CSR)

Matrix Graph

```
Gauss-Seidel Iteration Space

\[
\text{do iter = 1, } T \\
\text{do i = 1, } R \\
\hspace{2em} u(i) = f(i) \\
\hspace{2em} \text{do p = ia(i), } \text{ia(i+1) - 1} \\
\hspace{4em} j = ja(p) \\
\hspace{4em} \text{if (j ! = i)} \\
\hspace{6em} u(i) -= a(p) \times u(ja(p)) \\
\hspace{4em} \text{else} \\
\hspace{6em} \text{diag = a(p)} \\
\hspace{6em} \text{endif} \\
\hspace{2em} \text{enddo} \\
\hspace{2em} \text{endo} \\
\hspace{2em} u(i) = u(i) / \text{diag} \\
\hspace{2em} \text{enddo} \\
\hspace{2em} \text{endo}
\]
Performance Improvement Opportunities

- Intra-iteration reuse
- Inter-iteration reuse
- Parallelism
Turn Data Reuse into Data Locality

**Spatial locality** occurs when memory locations mapped to the same cache-line are used before the cache line is evicted.

**Temporal locality** occurs when the same memory location is reused before its cache line is evicted.
Full Sparse Tiling

- Breaks up computation into pieces
- Each piece has intra and inter-iteration data locality
- Some pieces can be executed in parallel
Talk Outline

- Overview
- Parallelizing Jacobi with the owner computes method or full sparse tiling
- Increasing the parallelism in full sparse tile schedules
- Extra work needed to handle Gauss-Seidel
- Experimental Results
Jacobi: similar yet simpler

- data dependences not known until runtime
- no intra-iteration dependences
- i loop is a reduction, therefore parallelizable

```
  do iter = 1, T
    do i = 1, R
      u(i) = f(i)
      do p = ia(i), ia(i+ 1) - 1
        j = ja(p)
        if (j != i) then
          u(i) -= a(p)*tmp (j)
        else
          diag = a(p)
        endif
      enddo
      u(i) = u(i)/diag
    enddo
  enddo
  do i = 1, R
    temp(i) = u(i)
  enddo
  do p = ia(i), ia(i+ 1) - 1
    do iter = 1, T
      do i = 1, R
        u(i) = f(i)
        j = ja(p)
        if (j != i) then
          u(i) -= a(p)*tmp (j)
        else
          ...   endif
      enddo
      u(i) = u(i)/diag
    enddo
  enddo
```
Parallelize with Owner Computes Method

At each convergence iteration ...

- each partition receives data from previous convergence iteration
- each partition executes in parallel
- each partition sends data
Locality in Owner Computes Method

- Intra-iteration locality
  - schedule by sub-part
  - reorder data for consecutive order in sub-parts

- NO Inter-iteration locality, main partitions don’t fit in cache
Jacobi Iteration Space
Parallelize with Full Sparse Tiling

- Create seed partitioning at $iter = 2$
Parallelize with Full Sparse Tiling

- Create seed partitioning at \( \text{iter} = 2 \)
- Grow tiles to \( \text{iter} = 1 \) based on ordering of partitions
Parallelize with Full Sparse Tiling

- Create seed partitioning at $iter = 2$
- Grow tiles to $iter = 1$ based on ordering of partitions
Parallelize with Full Sparse Tiling

- Create seed partitioning at $\text{iter} = 2$
- Grow tiles to $\text{iter} = 1$ based on ordering of partitions
Locality in Full Sparse Tiled Jacobi

- **Intra-iteration** locality achieved by consecutively ordering matrix graph nodes within a seed partition

- **Inter-iteration** locality achieved with tile-by-tile execution
Parallelism in Full Sparse Tiled Jacobi

Average parallelism = (# tiles) / (# tiles in critical path)

= 6 / 5 = 1.2
How can we increase parallelism between tiles?

- Order that tile growth is performed matters
- Best is to first grow tiles whose seed partitions are not adjacent
Improving Average Parallelism Using Coloring

- Create a partition graph
Improving Average Parallelism Using Coloring

- Create a partition graph
- Color the partition graph
Improving Average Parallelism Using Coloring

- Create a partition graph
- Color the partition graph
- Renumber partitions consecutively by color
Grow Using New Partition Order

- Renumber the seed partition cells based on coloring
- Grow tiles using new ordering
- Notice that tiles 0 and 1 may be executed in parallel
Re-grow Using New Partition Order

- Renumber the seed partition cells based on coloring
- Grow tiles using new ordering
- Notice that tiles 0 and 1 may be executed in parallel
- Tiles 4 and 5 may also be executed in parallel
Average Parallelism is Improved

Average parallelism = (# tiles) / (# tiles in critical path)

= 6 / 3 = 2
Improvement with Real Matrix Graphs

- **2D Bar**: R=74,926, NZ=1,037,676
- **3D Bar**: R=122,061, NZ=4,828,779
- **Sphere**: R=154,938, NZ=11,508,390
- **Pipe**: R=381,120, NZ=15,300,288

Graph Coloring Partition Numbering

Average Parallelism

- numtile = 14
- numtile = 58
- numtile = 135
- numtile = 183
Gauss-Seidel

- Loop carried dependences within convergence iteration as well as between them
- Dependences depend on the ordering of the nodes
- Nodes can be reordered apriori
Owner Computes Method
Nodal Gauss-Seidel [Adams 2001]

- Renumbers nodes (rows/columns in sparse matrix)
- Make data dependences between main partitions consistent
Owner Computes Method
Nodal Gauss-Seidel [Adams 2001]

• Renumbers nodes (rows/columns in sparse matrix)
• Make data dependences between main partitions consistent
• Subpartition for better parallelism
Full Sparse Tiled Gauss-Seidel

- Tile growth creates and maintains a partial ordering between nodes in matrix graph
- Reorder nodes so that partial ordering is maintained
Experimental Methodology

- Baseline is Gauss-Seidel implemented CSR and uses provided ordering
- **Owner computes implementation**
  - Partition matrix graph into equal-sized cells for each processor
  - Sub-partition and reorder on each processor for intra-iteration locality
  - Violate intra-iteration dependences for an idealistic parallel efficiency
- **Full sparse tiling implementation**
## Overhead of Full Sparse Tiling

### Blue Horizon, Gauss-Seidel with numiter=2, Parallel Reschedule

<table>
<thead>
<tr>
<th>Input Matrix</th>
<th>Overhead (sec)</th>
<th>Savings/Execution (sec)</th>
<th>Break Even</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>n=2</td>
<td>n=4</td>
<td>n=8</td>
</tr>
<tr>
<td>Matrix9</td>
<td>2.03</td>
<td>0.02</td>
<td>0.03</td>
</tr>
<tr>
<td>Matrix12</td>
<td>13.69</td>
<td>0.14</td>
<td>0.20</td>
</tr>
<tr>
<td>Sphere150K</td>
<td>31.41</td>
<td>0.17</td>
<td>0.26</td>
</tr>
<tr>
<td>PipeOT15mill</td>
<td>48.28</td>
<td>0.39</td>
<td>0.52</td>
</tr>
<tr>
<td>Wing903K</td>
<td>116.86</td>
<td>0.65</td>
<td>0.96</td>
</tr>
</tbody>
</table>

### Percentage Time of Overhead

<table>
<thead>
<tr>
<th>Input Matrix</th>
<th>Partitioning</th>
<th>Data Remapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix9</td>
<td>79%</td>
<td>14%</td>
</tr>
<tr>
<td>Matrix12</td>
<td>72%</td>
<td>13%</td>
</tr>
<tr>
<td>Sphere150K</td>
<td>68%</td>
<td>17%</td>
</tr>
<tr>
<td>PipeOT15mill</td>
<td>81%</td>
<td>10%</td>
</tr>
<tr>
<td>Wing903K</td>
<td>84%</td>
<td>10%</td>
</tr>
</tbody>
</table>
Experimental Results on IBM Blue Horizon

Blue Horizon, GS numiter=2

Raw Speedup

Sphere
  R=154,938
NZ=11,508,390

Pipe
  R=381,120
NZ=15,300,288

Wing
  R=924,672
NZ=38,360,266

Avg Par = 7.5
Avg Par = 30.5
Avg Par = 56.6
Experimental Results on Sun Ultra

Ultra, GS numiter = 2

- Sphere
  - R = 154,938
  - NZ = 11,508,390
- Pipe
  - R = 381,120
  - NZ = 15,300,288
- Wing
  - R = 924,672
  - NZ = 38,360,266

Avg Par = 9.3
Avg Par = 45.6
Avg Par = 90.5
Conclusions

• Full sparse tiling exploits parallelism, intra-iteration data reuse, and inter-iteration data reuse

• Coloring the partition graph to number seed partitions improves the average parallelism in the tile dependence graph

• On shared memory processors, full sparse tiling can outperform owner-computes methods when enough parallelism is present
Future Work

• Develop distributed memory execution framework for full sparse tiling

• Scaling studies for distributed memory execution

• Apply to other benchmarks and kernels
  - Jacobi
  - Moldyn
  - Molecular dynamics benchmark from Cornell

• Automate generation of inspectors and executors
PETSc - Portable Extensible Toolkit for Scientific Computing

- Parallel Partial Differential Equations (PDE) solvers using Object Oriented Programming (OOP)
- Used in many applications: CFD, Optimization, Biology, Finite Element Analysis, etc.
- Software architecture contains many aspects
  - Usage levels: beginner, intermediate, etc.
  - Many Krylov methods, preconditioners, and sparse matrix formats
  - Profiling, logging, options, etc.
Sparse Tiling Extension to PETSc (STPetsc)

• How should run-time reordering transformations be incorporated in existing domain-specific libraries?

• Interface
  - Beginner: Call MatUseST_SeqAIJ(A)
  - Intermediate: MatSparseTilingCreate(), etc.

• Implementation
  - Sparse tiling *inspector* occurs upon first call to SOR or before SLESSolve
  - SOR for SeqAIJ (CSR) matrix format is transformed into a sparse tiling *executor*
Cone Matrix on Pentium 4 2GHz
(N=22,032, NZ=1,433,068 )
Data Reordering Affects Convergence Cone Matrix

- Original
- Data Reorder Only in SOR
- Data Reorder Only before Solve
- Full Sparse Tile in SOR
- Full Sparse Tile before Solve

Time per Iteration

Iterations to Converge

Total Execution Time (setup+solve)
Sphere Matrix on Pentium 4 2GHz
(N=154,938, NZ=11,508,390)

Total Execution Time (setup+solve)
- Original
- Data Reorder Only in SOR
- Data Reorder Only before Solve
- Full Sparse Tile in SOR
- Full Sparse Tile before Solve

SOR iterations
0
100
200
300
400
500
600
700
800
900
1000

Setup Time
Data Reordering Affects Convergence

Sphere Matrix

- **Total Execution Time (setup+solve)**
  - Original
  - Data Reorder Only in SOR
  - Data Reorder Only before Solve
  - Full Sparse Tile in SOR
  - Full Sparse Tile before Solve

- **Time per Iteration**
- **Iterations to Converge**
Framework Elements
[Kelly & Pugh 95] [Pugh & Wonnacott 95]

Data Mapping

Compile-time: \( M_{I \rightarrow Z} = \{ [i] \rightarrow [x(i)] | 0 \leq i \leq 7 \} \)

Run-time: \[
\begin{array}{cccccccc}
I & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
Z & a & b & c & d & e & f & g & h \\
\end{array}
\]

Data Dependences

Compile-time: \( D_{I \rightarrow I} = \{ [i] \rightarrow [j] | (0 \leq i < j \leq 7) \land (i = b(j) \lor j = b(i)) \} \)

Run-time: \[
\begin{array}{cccccccc}
I & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\end{array}
\]
Key Insights for Composing Run-time Reordering Transformations

- The inspectors *traverse* the data mappings and/or the data dependences

- We can *express* how the data mappings and data dependences will change

- Subsequent inspectors *traverse the new* data mappings and data dependences
Data Mappings and Data Dependences for MOLDYN

for ts=1,T
  for i=1,N
    Z[i] = ...
  endfor
  for j=1,M
    ... = Z[l[j]]
    ... = Z[r[j]]
  endfor
endfor

Data Mapping for i loop

\[ M_{I_0 \rightarrow Z_0} = \{ [i] \rightarrow [i] \} \]

Data Mapping for j loop

\[ M_{J_0 \rightarrow Z_0} = \{ [j] \rightarrow [i] \mid (i = l(j)) \vee (i = r(j)) \} \]

Data Dependences
between i and j loop

\[ D_{I_0 \rightarrow J_0} = \{ [i] \rightarrow [j] \mid (i = l(j)) \vee (i = r(j)) \} \]
Composing the Inspector at Compile-time for MOLDYN

1. Traverse $M_{J_0 \rightarrow Z_0}$ to generate data reordering function $\sigma$

$$M_{J_0 \rightarrow Z_0} = \{[j] \rightarrow [i] \mid (i = l(j)) \vee (i = r(j))\}$$

$$R_{Z_0 \rightarrow Z_1} = T_{I_0 \rightarrow I_1} = \{[i] \rightarrow [\sigma(i)]\}$$

2. Traverse $M_{J_0 \rightarrow Z_1}$ to generate iteration reordering function $\delta$

$$M_{J_0 \rightarrow Z_1} = \{[j] \rightarrow [\sigma(i)] \mid (i = l(j)) \vee (i = r(j))\}$$

$$T_{J_0 \rightarrow J_1} = \{[j] \rightarrow [\delta(j)]\}$$

3. Full sparse tile by traversing $D_{I_1 \rightarrow J_1}$ to generate tiling $\theta$

$$D_{I_1 \rightarrow J_1} = \{[\sigma(i)] \rightarrow [\delta(j)] \mid (i = l(j)) \vee (i = r(j))\}$$

$$T_{I_1 \rightarrow I_2} = \{[i_1] \rightarrow [\theta(1, i_1), 1, i_1]\}$$

$$T_{J_1 \rightarrow J_2} = \{[j_1] \rightarrow [\theta(2, j_1), 2, j_1]\}$$