RBF Interpolation for Mesh Deformation

Patrick Schiffmann

AVL List GmbH, Hans-List-Platz 1, Graz

07.03.2017 - HPC I
Outline

Problem
  Mesh Deformation
  Interpolation with Radial Basis Functions

Initial Investigation

Optimisations

Summary
Problem

Applications

▶ Mesh Movement
▶ Boundary Layer Generation

Why RBF?

▶ Higher mesh quality than other methods
▶ Allows bigger step sizes

Why investigate HPC?

▶ High computational cost
▶ Not feasible using serial CPU
Example

Figure: Cut through a computational mesh, after and before deformation.
RBF Interpolation for Deformation

We have

- a computational mesh, treated as point cloud in $\mathcal{R}^3$
- a subset of points $\mathcal{X} = \{x_i\}_{i=1}^N$
- associated function values $f_i = f(x_i)$ for these points

We want

- Interpoland $s$ for $f$: $s|_\mathcal{X} = f|_\mathcal{X}$
- $s(x) = \sum_{i=1}^N \lambda_i \phi(\|x - x_i\|) + p(x)$

Specific case, multiquadric biharmonics

- $\phi(r) = \sqrt{r^2 + c^2}$, $c \in \mathcal{R}$
- $p(x) = c$
Linear System

Requiring the interpolation condition in all given points and demanding a side condition on the coefficients of the polynomial term leads to a system of linear equations for the determination of the coefficients $\vec{\lambda}$ and $\vec{\pi}$:

$$
\sum_{i=1}^{N} \lambda_{i} \phi(\|x_{i} - x_{k}\|) + \sum_{j=1}^{M} \pi_{j} p_{j}(x_{k}) = f(x_{k}), \quad 1 \leq k \leq N,
$$

$$
\sum_{i=1}^{N} \lambda_{i} p_{l}(x_{i}) = 0, \quad 1 \leq l \leq M,
$$

or, in short notation

$$
\begin{pmatrix}
\Phi & \Pi \\
\Pi^{\top} & 0
\end{pmatrix}
\begin{pmatrix}
\vec{\lambda} \\
\vec{\pi}
\end{pmatrix}
=
\begin{pmatrix}
\vec{f} \\
0
\end{pmatrix}.
$$
CPU

Shared Memory - OpenMP

- Max. Speed-up 5x reached on 16 cores

Hybrid - MPI & OpenMP

- Wrapper around Shared Memory code
- Theoretically less efficient algorithm
- Max. Speed-up similar
GPU

- GPU far outperforms CPU version
- Results are not identical numerically

Table: Performance Comparison between Serial and GPU Code for 90k BC nodes

<table>
<thead>
<tr>
<th></th>
<th>CPU [s]</th>
<th>GPU [s]</th>
<th>CPU/GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>brute force</td>
<td>3.44</td>
<td>0.03</td>
<td>105.08</td>
</tr>
<tr>
<td>FMM setup</td>
<td>0.16</td>
<td>0.03</td>
<td>4.75</td>
</tr>
<tr>
<td>FMM eval</td>
<td>1.44</td>
<td>0.02</td>
<td>68.03</td>
</tr>
</tbody>
</table>
Optimisations

Trivial
- New compilers
- Full optimization flags
- Reading -opt-reports

Non-Trivial
- Explicit vectorisation
- Algorithmic changes exposing more parallelism
- Proper use of C++ qualifiers (const, static) and templates
Explicit Vectorisation

- Used Agner Fog’s C++ Vectorisation Class Library (VCL)
- Easy to use, GPLv3 License
- Speed-Up ideal when using work arrays, memory bound when used with OOP style data structures

```cpp
Vec4d x_(coords[i_N][0], coords[i_N+1][0], ...);
Vec4d y_(coords[i_N][1], coords[i_N+1][1], ...);
Vec4d z_(coords[i_N][2], coords[i_N+1][2], ...);

Vec4d xfrac = x_ / z_;  // Vector division
Vec4d x_norm_square = (x_*x_) + (y_*y_) + (z_*z_);  // Sum of squares
Vec4d x_norm = sqrt(x_norm_square);  // Square root
Vec4d inv_x_norm_square = 1. / x_norm_square;  // Inverse
```
Results - Strong Scaling Intel

![Graph showing speedup and time for different core counts and data sizes.](image)
## Results - Intel vs ARM

### Intel Xeon system

<table>
<thead>
<tr>
<th>cores $\rightarrow$</th>
<th>sequential</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{14}$</td>
<td></td>
<td>2.69E+0</td>
<td>2.71E+0</td>
<td>1.42E+0</td>
<td>7.74E-1</td>
<td>4.75E-1</td>
</tr>
<tr>
<td>$2^{16}$</td>
<td></td>
<td>1.53E+1</td>
<td>2.11E+1</td>
<td>1.10E+1</td>
<td>6.38E+0</td>
<td>4.08E+0</td>
</tr>
<tr>
<td>$2^{18}$</td>
<td></td>
<td></td>
<td>1.37E+2</td>
<td>7.19E+1</td>
<td>3.82E+1</td>
<td>2.47E+1</td>
</tr>
<tr>
<td>$2^{20}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3.07E+2</td>
<td>1.87E+2</td>
</tr>
</tbody>
</table>

### ARM system

<table>
<thead>
<tr>
<th>cores $\rightarrow$</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>24</th>
<th>32</th>
<th>48</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{14}$</td>
<td>2.64E+0</td>
<td>1.82E+0</td>
<td>1.66E+0</td>
<td>1.52E+0</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>$2^{16}$</td>
<td>1.79E+1</td>
<td>1.14E+1</td>
<td>7.38E+0</td>
<td>6.79E+0</td>
<td>8.17E+0</td>
<td>7.58E+0</td>
</tr>
<tr>
<td>$2^{18}$</td>
<td>1.37E+2</td>
<td>7.18E+1</td>
<td>3.49E+1</td>
<td>3.25E+1</td>
<td>3.50E+1</td>
<td>3.20E+1</td>
</tr>
<tr>
<td>$2^{20}$</td>
<td>–</td>
<td>–</td>
<td>1.59E+2</td>
<td>1.30E+2</td>
<td>1.17E+2</td>
<td>9.66E+1</td>
</tr>
</tbody>
</table>

**Table:** Time per iteration on the two test systems.
Summary

▶ Algorithmic changes were required for scalability
▶ Code is very general, hindering performance
▶ There is plenty of room for node-level optimisations

Outlook

▶ Unify everything to one code base
▶ Use HPC friendly data structures
▶ Tune to specific test hardware
▶ Fully use heterogenous systems?