CS 294-73
Software Engineering for Scientific Computing

Homework Assignment 5
(and a little more)
Multigrid

\[ vcycle(\phi, \rho) \]
\[
\{ \\
\phi := \phi + \lambda (L(\phi) - \rho) \quad p \text{ times} \\
\text{if } (\text{level} > 0) \\
\{ \\
\mathcal{R} = \rho - L(\phi) \\
\mathcal{R}_c = A(\mathcal{R}) \\
\delta : B_c \to \mathbb{R} , \delta = 0 \\
vcycle(\delta, \mathcal{R}_c) \\
\phi := \phi + \mathcal{I}(\delta) \\
\phi := \phi + \lambda \ast (L(\phi) - \rho) \quad p \text{ times} \\
\} \\
\text{else} \\
\{ \\
\phi := \phi + \lambda \ast (L(\phi) - \rho) \quad p_B \text{ times} \\
\} \\
\} \\
\]

At the top level, iterate until residual is reduced by some large factor.
Baseline implementation of Residual

Multigrid::residual(...)  
{  
...  
res.setVal(0.);  
for (Point pt=bx.getLowCorner(); bx.notDone(pt); bx.increment(pt))  
{  
  for (int dir = 0; dir < DIM ; dir++)  
  {  
    Point edir = getUnitv(dir);  
    res[pt] += (a_phi[pt + edir] + a_phi[pt-edir]);  
  }  
  res[pt] = res[pt]*dimqtr + .5*a_phi[pt];  
}  
for (Point pt=bx.getLowCorner(); bx.notDone(pt); bx.increment(pt))  
{  
a_phi[pt] = res[pt] - m_lambda*a_rhs[pt];  
}
Getting timer output

```plaintext
> setenv CH_TIMER
> printenv
...
CH_TIMER =
> ./mg2D.exe < infile
...
total flops = 93148718
And we look at time.table

[2]mg 0.36704 13
  100.0% 0.3670 13 vcycle [3]
  100.0% Total

[3]vcycle 0.36703 13
  64.1% 0.2354 26 relax [4]
  24.8% 0.0910 13 vcycle [5]
  5.5% 0.0203 13 residual [10]
  2.7% 0.0101 13 avgdown [12]
  2.7% 0.0099 13 fineInterp [13]
  99.9% Total

Is this good or bad? 93148718 / .367 = 254 Mflops (not good).
```
(1) Good news: looks like we reduce the run time by 6X.
(2) We haven’t gotten the big pieces; Furthermore, even after we get relax, avgdown and fineInterp start to look relatively large.
Replace Baseline Implementation with Pencils

---

[2]mg 0.06940 13
100.0% 0.0694 13 vcycle [3]
100.0% Total

---

[3]vcycle 0.06940 13
39.4% 0.0274 26 relax [4] (vs. 0.2405)
26.1% 0.0181 13 vcycle [6]
15.6% 0.0108 13 fineInterp [7]
14.5% 0.0101 13 avgdown [9]
4.0% 0.0028 13 residual [14] (vs. 0.0203)
99.6% Total

---

(1) avgdown and fineInterp are now 20% of the time, so we probably want to get them, if it is not too painful.
Baseline implementation of avgdown

Multigrid::residual(...)
{
    ...
    DBox bxKernel(getZeros(),getOnes());
    double normalize = 1.0/bxKernel.sizeOf();
    for (Point ptc = bxc.getLowCorner(); bxc.notDone(ptc);
         bxc.increment(ptc))
    {
        for (Point ptsten = getZeros(); bxKernel.notDone(ptsten);
             bxKernel.increment(ptsten))
        {
            a_resc[ptc] += a_res[ptc*2 + ptsten];
        }
        a_resc[ptc]*=normalize;
    }
}
Timing subsections

You need to know how you are doing at the small scale. You can time subsections, but you need to count the flops as well. Modify the flop counters, and add member data

Multigrid::vCycle(...) 
{
    ... 
    pointRelax(a_phi,a_rhs,m_preRelax);
    m_flops += m_preRelax*m_box.sizeOf()*(2*DIM + 5);
    if (m_level > 0)
    {
        residual(m_res,a_phi,a_rhs);
        m_flops += m_box.sizeOf()*(2*DIM + 4);
        avgDown(m_resc,m_res);
        m_flops += m_box.sizeOf();
        m_delta.setVal(0.);
        m_coarsePtr->vCycle(m_delta,m_resc);
        fineInterp(a_phi,m_delta);
        m_flops += m_box.sizeOf();
        pointRelax(a_phi,a_rhs,m_postRelax);
        m_flops += m_postRelax*m_box.sizeOf()*(3*DIM + 3); ...
    
}
Timing subsections

You need to know how you are doing at the small scale. You can time subsections, but you need to count the flops as well. Modify the flop counters, and add member data

Multigrid::vCycle(...)
{
    ...
    pointRelax(a_phi,a_rhs,m_preRelax);
    m_flops += m_preRelax*m_box.sizeOf()*(2*DIM + 5);
    if (m_level > 0)
    {
        residual(m_res,a_phi,a_rhs);
        m_flops += m_box.sizeOf()*(2*DIM + 4);
        avgDown(m_resc,m_res);
        m_flops += m_box.sizeOf();
        m_delta.setVal(0.);
        m_coarsePtr->vCycle(m_delta,m_resc);
        fineInterp(a_phi,m_delta);
        m_flops += m_box.sizeOf();
        pointRelax(a_phi,a_rhs,m_postRelax);
        m_flops += m_postRelax*m_box.sizeOf()*(3*DIM + 3);  ...
}
Vectorization

Using clang:
CFLAGS += -Rpass=loop-vectorize
If you want to know what loops are failing to vectorize, and why,
CFLAGS += -Rpass-analysis=loop-vectorize

Using g++:
CFLAGS+=-ftree-vectorizer-verbose=2
To turn off vectorization:
CFLAGS += -fno-vectorize
(However, g++ is more pessimistic about the value of vectorization).
Vectorization

Using clang:
CFLAGS += -Rpass=loop-vectorize - tells you what loops vectorize
If you want to know what loops are failing to vectorize, and why,
CFLAGS += -Rpass-analysis=loop-vectorize

Using g++:
CFLAGS+=-ftree-vectorizer-verbose=2
To turn off vectorization:
CFLAGS += -fno-vectorize
(However, g++ is more pessimistic about the value of vectorization).
Pencils only work if unambiguous

double* a, b, c, d;
...
for (int k = 0; k < bar.size(); k++)
{
    a[k] = b[k] * c[k] + d[2*k];
}
Fails. Instead, use
double* a, b, c;
...
int high = bar.size();
for (int k = 0; k < high; k++)
{
    a[k] = b[k] * c[k] + d[2*k];
}

Loop bodies have to have obvious constant-stride access, fixed loop limits.
If we replace double* a, b, c, d; with vector a, b, c, d;
will the compiler recognize the unit stride access?
An Embedded Domain-Specific Language

\[ \text{vcycle}(\phi, \rho) \]
\[
\{ \\
\phi := \phi + \lambda(L(\phi) - \rho) \text{ p times} \\
\text{if (level > 0)} \\
\{ \\
\mathcal{R} = \rho - L(\phi) \\
\mathcal{R}_c = \mathcal{A}(\mathcal{R}) \\
\delta : B_c \rightarrow \mathbb{R}, \delta = 0 \\
\text{vcycle}(\delta, \mathcal{R}_c) \\
\phi := \phi + \mathcal{I}(\delta) \\
\phi := \phi + \lambda * (L(\phi) - \rho) \text{ p times} \\
\} \\
\text{else} \\
\{ \\
\phi := \phi + \lambda * (L(\phi) - \rho) \text{ p}_B \text{ times} \\
\} 
\]

At the level of \text{vcycle} our code looks like this, but inside it is messy, particularly when you start optimizing for performance.

Can we do better?

- Identify programming abstractions corresponding to the high-level mathematical abstractions.
- Implement as a library that curates the optimizations you do by hand.
- (apply compiler technologies to make it even better).
An Embedded Domain-Specific Language

At the level of vcycle our code looks like our pseudocode, but inside it is messy, particularly when you start optimizing for performance.

Can we do better?

• Identify programming abstractions as aggregate operations corresponding to the high-level mathematical abstractions.
• Implement as C++ classes that contain the optimizations you would otherwise do by hand. The idea is that the opportunities for obtaining high performance come from the mathematical structure of the algorithms.
• (apply compiler technologies to make it even better).

RectMDArray does half of this, i.e. defines high-level data structures. We'll now take a look at the other half of the RectMDArray infrastructure, which is a stencil language.

(Joint work with Brian Van Straalen, Chris Gebhart).
Stencil Operators

\[ \phi : B \rightarrow \mathbb{R}^N , \ B \text{ a Box} \]

\[ L(\phi)_i = \sum_{s \in S} a_s \phi_{i+s} , \ i \in B' \]

Can think of the operator L as an object that acts any RectMDArray.

\[ L = \sum_{s \in S} a_s S_s^s , \ S_s^s(\phi)_i = \phi_{i+s} \]

Stencil operators have their own algebra: you can add them, compose them, multiply them by scalars, without knowing the details of what they will be applied to.

\[ L_{1,2} = \sum_{s \in S} a_{s}^{1,2} S_s^s \]

\[ L_1 \circ L_2 = \sum_t ( \sum_{s+s' = t} a_s^1 a_{s'}^2 ) S_t^t \]
Stencil Operators

Stencil<double> m_Lap2nd;

m_Lap2nd =
    Stencil<double>(2.0*DIM)*Shift(getZeros());

for (int dir = 0; dir < DIM ; dir++)
{
    Point edir = getUnitv(dir);
    Stencil<double> plus = 1.0*Shift(edir);
    Stencil<double> minus = 1.0*Shift(edir*(-1));

    m_Lap2nd = m_Lap2nd + minus + plus;
}
Multigrid::residual(
    RectMDArray<double>& a_res,
    RectMDArray<double>& a_phi,
    RectMDArray<double>& a_rhs
)
{
    CH_TIMERS("residual");
    DBox bx = m_box;
    getGhost(a_phi);
    double hsqi = 1.0/(m_dx*m_dx);
    a_res |= m_Lap2nd(a_phi, bx, -hsqi);
    a_res += a_rhs;
};
Stencil Operators

Stencils can be strided, both on input and on output. Useful for coarsening / refinement.

\[
\mathcal{L}(\phi)_{ir^\text{dest} + q^\text{dest}} = \sum_s a_s \phi_{ir^\text{src} + s}, i \in \Gamma
\]
Arithmetic on RectMDArrays

\[ (f@U)_i \equiv f(U)_i , \ i \in B \]

forall(fOfU,U,f,bx); // f = f(U_in,U_out).
we can also use lambda calculus in C++11 to apply partially evaluated functions.
forall(fOfU,U,[dt](State& a,const State& b){return f(a,b,dt);}), bx);
Use lambdas to apply member functions of a class.
forall(W, U,
[this](State& a, const State& b){return consToPrim(a,b);}, B_3);
How do we get performance?

In the apply function, we implement the pencil construction once.

```c
... 
for (int isten =0;isten < srcOffset.size(); isten++)
{
    int kpoffset = kbasep+srcOffset[isten];
    double coefpt = coef[isten];
    for (int k0 = 0;k0 < nptsDst;k0++)
    {
        const T& phival = (phi_ptr_i)[kpoffset+k0];
        T& lofphi = (lofphi_ptr_i)[kbasel+k0];
        lofphi+=coefpt*phival;
    }
}
...
```
How is the performance?

---

[2] mg 0.07775 13

100.0%  0.0777

13 vcycle [3]

100.0%  Total

---

[3] vcycle 0.07774 13 (compare to .347 baseline, .056 by hand).

58.5%  0.0455  26 relax [4]

28.5%  0.0222  13 vcycle [8]

6.4%   0.0050  13 residual [16]

3.4%   0.0027  13 fineInterp [24]

2.7%   0.0021  13 avgdown [30]

99.6%  Total

---

i.e. we get about 1.2 Gflops from stencil, vs. 1.7 Gflops by hand, and 268 Mflops baseline.

• This is respectable. If we want to get the last 40% on a few performance-critical sections, we can tune those by hand.

• There is another advantage to stencils, which is productivity. It is actually better than the baseline (and of the hand-tuned), from that standpoint.
Multigrid in a slide

// residual.
a_res |= m_Lap2nd(a_phi, bx, -hsqi);
a_res += a_rhs;

// pointRelax:
for (int iter = 0; iter < a_numIter; iter++)
{
    getGhost(a_phi);
    {lofphi |= Identity(a_rhs, m_box, -m_lambda);
     lofphi += m_Jacobi(a_phi, m_box);
     lofphi.copyTo(a_phi);}
}

// fineInterp:
for (Point ptsten = Point::getZeros();
    m_boxSten.notDone(ptsten); m_boxSten.increment(ptsten))
    {a_phi += m_Interp(ptsten, 0)(a_delta, bxc);}

// avgdown:
    DBox bxc = a_resc.getDBox();
a_resc |= m_avgDown(a_res, bxc);
Stencils can be curated in a library.

//List of Implemented Stencils

<table>
<thead>
<tr>
<th>STRING:</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identity:</td>
<td>Trivial identity calculation</td>
</tr>
<tr>
<td>FirstDeriv_2C[DIM]:</td>
<td>First Derivative, 2nd Order, Centered</td>
</tr>
<tr>
<td>FirstDeriv_2L[DIM]:</td>
<td>First Derivative, 2nd Order, Low Edge</td>
</tr>
<tr>
<td>FirstDeriv_2H[DIM]:</td>
<td>First Derivative, 2nd Order, High Edge</td>
</tr>
<tr>
<td>FirstDeriv_4C[DIM]:</td>
<td>First Derivative, 4th Order, Centered</td>
</tr>
<tr>
<td>FirstDeriv_4L[DIM]:</td>
<td>First Derivative, 4th Order, Low Edge</td>
</tr>
<tr>
<td>FirstDeriv_4H[DIM]:</td>
<td>First Derivative, 4th Order, High Edge</td>
</tr>
<tr>
<td>SecondDeriv_4C[DIM]:</td>
<td>Second Derivative, 4th Order, Centered</td>
</tr>
<tr>
<td>Laplacian_2:</td>
<td>Laplacian, 2nd Order</td>
</tr>
<tr>
<td>Laplacian_19:</td>
<td>Laplacian, 19 Point (3D)</td>
</tr>
<tr>
<td>Laplacian_27:</td>
<td>Laplacian, 27 Point (3D)</td>
</tr>
<tr>
<td>Laplacian_117:</td>
<td>Laplacian, 117 Point (3D)</td>
</tr>
<tr>
<td>EdgeToCell_4[DIM]:</td>
<td>Edge-&gt;Cell Interp, 4th Order</td>
</tr>
<tr>
<td>EdgeToCell_6[DIM]:</td>
<td>Edge-&gt;Cell Interp, 6th Order</td>
</tr>
<tr>
<td>CellToEdge_5L[DIM]:</td>
<td>Cell-&gt;Edge Interp, 5th Order, Low Edge</td>
</tr>
<tr>
<td>CellToEdge_5H[DIM]:</td>
<td>Cell-&gt;Edge Interp, 5th Order, High Edge</td>
</tr>
<tr>
<td>CellToEdge_7L[DIM]:</td>
<td>Cell-&gt;Edge Interp, 7th Order, Low Edge</td>
</tr>
<tr>
<td>CellToEdge_7H[DIM]:</td>
<td>Cell-&gt;Edge Interp, 7th Order, High Edge</td>
</tr>
<tr>
<td>CellToEdge_9L[DIM]:</td>
<td>Cell-&gt;Edge Interp, 9th Order, Low Edge</td>
</tr>
<tr>
<td>CellToEdge_9H[DIM]:</td>
<td>Cell-&gt;Edge Interp, 9th Order, High Edge</td>
</tr>
<tr>
<td>FluxDivergence[DIM]:</td>
<td>Flux Differencing</td>
</tr>
</tbody>
</table>
Stencils can be curated in a library.

```cpp
Multigrid::define(...) {
...
  m_Lap2nd = SLib::get("Laplacian_2");
  m_Jacobi = m_Lap2nd;
  m_Jacobi = (4.0*DIM)*Shift(Point::getZeros()) + m_Jacobi;
  m_Jacobi*=1.0/(4.0*DIM);
  m_avgDown = SLib::get("AvgDown2");
  m_boxSten=DBBox(Point::getZeros(),Point::getOnes());
  m_Interp.define(m_boxSten);
  for (Point pt = Point::getZeros(); m_boxSten.notDone(pt); m_boxSten.increment(pt)) {
    m_Interp(pt,0) = (1.0)*Shift(Point::getZeros());
    m_Interp(pt,0).setDestRefratio(Point::getOnes()*2);
    m_Interp(pt,0).setDestShift(pt);
  }
...
```
Summary

- Small constant stride 1 access (e.g. stride 1), vectorization, make a big performance difference.
- You will find that doing by hand is drudgery.
- Higher-level abstractions can enable performance engineering by making them reuseable. Also allows you to improve performance without rewriting large amounts of application code.
- (Stencil library is available for use in structured grid projects, if anyone is feeling adventurous).