MAR 572
Geophysical Simulation

Wed/Fri 3:00-4:20 pm
Endeavour 158

Prof. Marat Khairoutdinov
Office hours: Endeavour 121; by email appt
Email: marat.khairoutdinov@stonybrook.edu
Class website: http://rossby.msrc.sunysb.edu/~marat/MAR572.html
login: MAR572; passwd: coolclouds

Recommended Textbooks

Grading
50% Homework
50% 2 exams & 1 midterm project (No final)

Grade conversion: A: >90, B+: 86-90, B: 76-85, C+: 70-75, C: <70

Homework Policy
Homework will be handed out weekly and is due in one week. After the due date, homework can be turned in for 50% credit. Homework can be discussed with other students; however, each student is expected to write the solutions independently.

Hands-on experience is the best way to learn numerical methods. Homework will involve writing simple programs and plotting the results. You will need to have access to computers with programming and graphing software. Knowledge of high-level compiled (e.g., Fortran (preferred), C) or scripting (e.g., IDL, Matlab) computer languages is required for this course. It is, however, up to you which programming language or graphing application to use.
MAR 572
Geophysical Simulation

Fundamentals of Finite-Difference Schemes
Definitions of consistence, convergence, and stability; First and second order derivatives; Construction of higher order approximations; Numerical solution of nonlinear equations.

Methods for Initial-Value Problems of Linear Partial Differential Equations
Linear computational stability analysis; Classification and canonical forms; Basic numerical schemes for advection and diffusion equations; Upstream and downstream biased schemes; Time-integration schemes; Time-splitting and directional splitting schemes; Implicit and explicit schemes; Numerical diffusion and dispersion; Extension to multiple dimensions; Grid systems.

Methods for Nonlinear Initial-Value Problems
Fourier representation of discrete fields; Nonlinear interaction and instability; Methods to eliminate nonlinear instability; Construction of conservation schemes; Monotonic and positive definite schemes; Barotropic vorticity model; the Arakawa Jacobian; Basic concepts of spectral methods; Semi-Lagrangian and finite-volume methods

Methods to Solve Elliptic Equations
Fourier method; Relaxation methods; Multi-grid methods; Tri-diagonal matrix solver

Data Analysis
Classical objective analysis; Statistical estimation; Maximum likelihood estimation; Least variance estimation; Kalman filtering; Statistical spatial interpolation; Variational analysis method; Adjoint models; Multivariant analysis
Top 500 supercomputers

Performance Development

- 1 Eflop/s
- 100 Pflop/s
- 10 Pflop/s
- 1 P flop/s
- 10 Tflop/s
- 1 Tflop/s
- 100 Gflop/s
- 10 Gflop/s
- 1 Gflop/s

- 224 PFlop/s
- 33.9 PFlop/s
- 96.6 TFlop/s

- 1.17 TFlop/s
- 59.7 GFlop/s

- SUM
- N=1
- N=500

- 400 MFlop/s

<table>
<thead>
<tr>
<th>#</th>
<th>Site</th>
<th>Manufacturer</th>
<th>Computer</th>
<th>Country</th>
<th>Cores</th>
<th>$R_{\text{max}}$ \text{(Piloops)}</th>
<th>Power \text{(MW)}</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>National University of Defense Technology</td>
<td>NUDT</td>
<td>Tianhe-2 NUDT TH-IVB-FEP, Xeon 12C 2.2GHz, IntelXeon Phi</td>
<td>China</td>
<td>3,120,000</td>
<td>33.9</td>
<td>17.8</td>
</tr>
<tr>
<td>2</td>
<td>Oak Ridge National Laboratory</td>
<td>Cray</td>
<td>Titan Cray XK7, Opteron 16C 2.2GHz, Gemini, NVIDIA K20x</td>
<td>USA</td>
<td>560,640</td>
<td>17.6</td>
<td>8.21</td>
</tr>
<tr>
<td>3</td>
<td>Lawrence Livermore National Laboratory</td>
<td>IBM</td>
<td>Sequoia BlueGene/Q, Power BQC 16C 1.6GHz, Custom</td>
<td>USA</td>
<td>1,572,864</td>
<td>17.2</td>
<td>7.89</td>
</tr>
<tr>
<td>4</td>
<td>RIKEN Advanced Institute for Computational Science</td>
<td>Fujitsu</td>
<td>K Computer SPARC64 VIIIfx 2.0GHz, Tofu Interconnect</td>
<td>Japan</td>
<td>795,024</td>
<td>10.5</td>
<td>12.7</td>
</tr>
<tr>
<td>5</td>
<td>Argonne National Laboratory</td>
<td>IBM</td>
<td>Mira BlueGene/Q, Power BQC 16C 1.6GHz, Custom</td>
<td>USA</td>
<td>786,432</td>
<td>8.59</td>
<td>3.95</td>
</tr>
<tr>
<td>6</td>
<td>Texas Advanced Computing Center/UT</td>
<td>Dell</td>
<td>Stampede PowerEdge C8220, Xeon E5 8C 2.7GHz, Intel Xeon Phi</td>
<td>USA</td>
<td>462,462</td>
<td>5.17</td>
<td>4.51</td>
</tr>
<tr>
<td>7</td>
<td>Forschungszentrum Juelich (FZJ)</td>
<td>IBM</td>
<td>JuQUEEN BlueGene/Q, Power BQC 16C 1.6GHz, Custom</td>
<td>Germany</td>
<td>458,752</td>
<td>5.01</td>
<td>2.30</td>
</tr>
<tr>
<td>8</td>
<td>Lawrence Livermore National Laboratory</td>
<td>IBM</td>
<td>Vulcan BlueGene/Q, Power BQC 16C 1.6GHz, Custom</td>
<td>USA</td>
<td>393,216</td>
<td>4.29</td>
<td>1.97</td>
</tr>
<tr>
<td>9</td>
<td>Leibniz Rechenzentrum</td>
<td>IBM</td>
<td>SuperMUC iDataPlex DX360M4, Xeon E5 8C 2.7GHz, InfiniBand FDR</td>
<td>Germany</td>
<td>147,456</td>
<td>2.90</td>
<td>3.52</td>
</tr>
<tr>
<td>10</td>
<td>National SuperComputer Center in Tianjin</td>
<td>NUDT</td>
<td>Tianhe-1A NUDT TH MPP, Xeon 6C, NVIDIA, FT-1000 8C</td>
<td>China</td>
<td>186,368</td>
<td>2.57</td>
<td>4.04</td>
</tr>
</tbody>
</table>
System for Atmospheric Modeling (SAM) Cloud-Resolving Model (CRM)

Momentum equations (anelastic)

\[
\frac{\partial u_i}{\partial t} = -\frac{1}{\rho} \frac{\partial}{\partial x_j} \left( \bar{\rho} u_i u_j + \tau_{ij} \right) - \frac{\partial}{\partial x_i} p' + \delta_{i3} B \\
+ \epsilon_{ij3} f (u_j - U_g) + \left( \frac{\partial u_i}{\partial t} \right)_{1.s.}, \tag{A1}
\]

\[
\frac{\partial \bar{\rho} u_i}{\partial x_i} = 0, \tag{A2}
\]

Scalar equations

\[
\frac{\partial h_L}{\partial t} = -\frac{1}{\rho} \frac{\partial}{\partial x_i} \left( \bar{\rho} u_i h_L + F_{h_L} \right) \\
- \frac{1}{\rho} \frac{\partial}{\partial z} (L_c P_r + L_s P_s + L_s P_g) \\
+ \left( \frac{\partial h_L}{\partial t} \right)_{rad} + \left( \frac{\partial h_L}{\partial t} \right)_{1.s.}, \tag{A3}
\]

\[
\frac{\partial q_r}{\partial t} = -\frac{1}{\rho} \frac{\partial}{\partial x_i} (\bar{\rho} u_i q_T + F_{q_T}) - \left( \frac{\partial q_r}{\partial t} \right)_{mic} \\
+ \left( \frac{\partial q_T}{\partial t} \right)_{1.s.}, \text{ and} \tag{A4}
\]

\[
\frac{\partial q_p}{\partial t} = -\frac{1}{\rho} \frac{\partial}{\partial x_i} (\bar{\rho} u_i q_p + F_{q_p}) \\
+ \frac{1}{\rho} \frac{\partial}{\partial z} (P_r + P_s + P_g) + \left( \frac{\partial q_p}{\partial t} \right)_{mic}. \tag{A5}
\]

\[
h_L = c_p T + g z - L_c (q_c + q_r) - L_s (q_i + q_s + q_g)
\]

\[
q_T = q_v + q_c + q_i
\]

Khairoutdinov and Randall (2003)
The Storage Hierarchy

- CPU Register
- Cache
  - Level 1
  - Level 2
- RAM
  - Physical RAM
  - Virtual Memory
- Storage Device Types
  - ROM/BIOS
  - Removable Drives
  - Network/Internet Storage
  - Hard Drive
- Input Sources
  - Keyboard
  - Mouse
  - Removable Media
  - Scanner/Camera/Mic/Video
  - Remote Source
  - Other Sources

Speed: Top to Bottom
Cost: Bottom to Top
Random Access Memory - RAM
(Main memory)

CPU speed is much faster than the transfer speed between RAM and CPU; hence, if all data were only in RAM, the CPU would be idle most of the time.
Cache/CPU speed is nearly the same as CPU speed;.
Data should be continuously prefetched/pipelined to cache which then supplies CPU as needed.
Hence, the data flow between RAM/Cache/CPU should be predicted.
The instructions for the data flow are usually supplied by a compiler.
But you need to help it too!

Cache
Can have levels. L1, L2, L3
Each subsequent level is faster (but smaller - expensive!) than the previous

Bottleneck (Slow)  Fast
Strategies for efficiency

- **Registers**: do maximum work at data already at registers before requesting new data

- **Cache**: maximum use of data already in cache; understand how data are stored in cache

- **Data locality**: priority access the data bits that already close to each other in memory before the bits that are far

- **Input/Output**: generally avoid at all costs; if must, do as much as possible at once rather than little bit at a time
Avoid Data Dependency

\[ x = y + b \]
\[ z = x + c \]

\[ \downarrow \]

\[ x = y + b \]
\[ z = y + b + c \]

Usually automatically done by a compiler when optimization is on.
Avoid explicit integer power

\[ x = y^{**3.0} \text{ expensive power-function may be called} \]

\[ \downarrow \]

\[ x = y \times y \times y \]

Can be automatically done by a ‘smart’ compiler
Avoid divisions

\[ x = y / 2. \]
\[ \downarrow \]
\[ x = 0.5 \times y \]

division can be much more expensive than multiplication

Can be automatically done by a ‘smart’ compiler
Reduce computation of common subexpressions

\[
\begin{align*}
x &= y \times (b / c) \\
z &= f \times (b / c)^{2.5}
\end{align*}
\]

Can be automatically done by a ‘smart’ compiler
Watch out for Loop Invariants

do i=1,n
   \( x(i) = y(i) + z(i) \cdot a/c(i)/b \) \hspace{1cm} a/b does not depend on \( i \)
   \( h(n) = f \cdot a \) \hspace{1cm} h(n) does not depend on \( i \)
end do

tmp = a/b

do i=1,n
   \( x(i) = y(i) + z(i) \cdot \text{tmp}/c(i) \)
end do

h(n) = f \cdot a
Look for conditions that dependent on loop index

Before

\[
\begin{align*}
\text{do } i &= 1, nx \\
\text{do } j &= 1, ny \\
\quad \text{if } (x(j) > 0) \text{ then} \\
\quad &\quad y(i, j) = x(i) + z(i, j) \times c(i) \\
\quad \text{else} \\
\quad &\quad y(i, j) = z(i, j) \times b(i) \\
\text{end if} \\
\text{end do} \\
\text{end do}
\end{align*}
\]

After

\[
\begin{align*}
\text{do } j &= 1, ny \\
\quad \text{if } (x(j) > 0) \text{ then} \\
\quad &\quad \text{do } i = 1, nx \\
\quad &\quad \quad y(i, j) = x(i) + z(i, j) \times c(i) \\
\quad &\quad \text{end do} \\
\quad \text{else} \\
\quad &\quad \text{do } i = 1, nx \\
\quad &\quad \quad y(i, j) = z(i, j) \times b(i) \\
\quad \text{end do} \\
\quad \text{end if} \\
\text{end do}
\end{align*}
\]
Get boundary conditions outside the loop

\begin{align*}
\text{do } i &= 1, n \\
 & \text{ if } (i == 1 \text{ or } i == n) \text{ then} \\
 & \quad x(i) = y(i) \\
 & \text{ else} \\
 & \quad x(i) = y(i) \cdot d(i) \\
 & \text{ end if} \\
\text{end do}
\end{align*}

Before

After

\begin{align*}
& x(1) = y(1) \\
& \text{do } i = 1, n \\
& \quad x(i) = y(i) \cdot d(i) \\
& \text{end do} \\
& x(n) = y(n)
\end{align*}
Index Splitting

**Before**

```plaintext
do i=1,n
  if (i<m) then
    x(i) = y(i)*c(i)
  else
    x(i) = y(i)*d(i)
  end if
end do
```

**After**

```plaintext
do i=1,m-1
  x(i) = y(i)*c(i)
end do

doi=m,n
  x(i) = y(i)*d(i)
end do
```
Loop order exchange

Make ‘slower’ changing loop-index to be in outer loop,
fast index - inner loop

Before

```
    do i=1,nx
        do j =1,ny
            x(i,j) = y(i,j)+1
        end do
    end do
```

After

```
    do j=1,ny
        do i =1,nx
            x(i,j) = y(i,j)+1
        end do
    end do
```

In Fortran, array a(3,3) is stored as:

- a(1,1) a(2,1) a(3,1) a(1,2) a(2,2) a(3,2) a(1,3) a(2,3) a(3,3)

In C the order is reversed:

- a(1,1) a(1,2) a(1,3) a(2,1) a(2,2) a(2,3) a(3,1) a(3,2) a(3,3)
Loop fusion  
(register reuse)  
Only if all arrays in the loop fit in cache (small)

Before

```plaintext
do i=1,n
  x(i) = y(i)+1
end do
do i=1,n
  a(i) = x(i)+c
end do
do i=1,n
  d(i) = a(i)+y(i)
end do
```

After

```plaintext
do i=1,n
  x(i) = y(i)+1
end do
do i=1,n
  a(i) = x(i)+c
  d(i) = a(i)+y(i)
end do
```
Loop fission
(effective cache use)

Only if all arrays are large, so all don’t fit in cache

Before

\[
\begin{align*}
\text{do } i=1, n \\
x(i) &= y(i) + 1 \\
a(i) &= x(i) + c \\
d(i) &= a(i) + y(i)
\end{align*}
\]

end do

After

\[
\begin{align*}
\text{do } i=1, n \\
x(i) &= y(i) + 1 \\
a(i) &= x(i) + c \\
d(i) &= a(i) + y(i)
\end{align*}
\]
Avoid loop index dependencies

Before

\[
\text{do } i=1,n \\
\quad x(i) = x(i-1) \times a(i) \\
\quad b(i) = x(i) \times c(i) \\
\text{end do}
\]

After

\[
\text{do } i=0,n-1 \\
\quad y(i) = x(i) \times a(i+1) \\
\text{end do} \\
\text{do } i=1,n \\
\quad x(i) = y(i) \times c(i) \\
\text{end do}
\]
avoid if’s when can

if (x >= 1.) then
    x = 1.
elseif (x <=0.) then
    x = 0.
end if

x=min(1.,max(0.,x))
avoid if’s when can

if (u(i) >= 0.) then
    x(i) = u(i)*(x(i)-x(i-1))
else
    x(i) = u(i)*(x(i+1)-x(i))
end if

v=max(0.,u(i)) * (x(i)-x(i-1))+ &
min(0.,u(i)) * (x(i+1)-x(i))

Before

After
Never call a subroutine/function within large loop

Before

```
do j=1,ny
  do i=1,nx
    y = fun(x(i,j))
  end do
end do
```

```real function fun (x)
  x = x*cos(x)
end function fun```

After

```
do j=1,ny
  do i=1,nx
    y = x(i,j)*cos(x(i,j))
  end do
end do
```

Inlining
Steps in code development:
Never call a subroutine/function within large loop

Before

```
    do j=1,ny
        do i=1,nx
            ...
            call micro()
            ...
        end do
    end do
```

subroutine micro()
    ...
    do j=1,ny
        do i=1,nx
            ...
        end do
    end do
    ...
end subroutine micro

After

```
    subroutine micro()
        ...
        do j=1,ny
            do i=1,nx
                ...
            end do
        end do
    ...
end subroutine micro
```
Some advise

- New code should go through initial test compiling with optimization off, uninitialized memory and array bound checks on;
- If the program fails after ported to another system or after not using the code for awhile, try running first with optimization off;
- Always try to use maximum compiler optimization; but make sure the results are similar (not the same) with lower optimization
- Use comments, sensible names for variables, indenting;
- One file per subroutine/module;
- Never use ‘magic numbers’.