High Performance Computing Lab Exercises

(Make sense of the theory!)

Rubin H Landau
With
Sally Haerer and Scott Clark

Computational Physics for Undergraduates
BS Degree Program: Oregon State University

“Engaging People in Cyber Infrastructure”
Support by EPICS/NSF & OSU
Programming for Virtual Memory

- Avoid page faults ($$)
- Avoid resource conflicts (|| I/O)
  - hurt other users
  - hurt yourself

VM Programming Rules

1. Worry only if use much memory
2. Look at entire program (global optimization)
3. Make successive calculations on data sub sets that fit in RAM
4. Avoid simultaneous calculations on same data set \([M^2/\text{Det}(M)]\)
5. Group together in memory data used together \((\neq \sum_i M_{ii})\)
Comparison: Java vs F90 & C

- **Java**
  - most universal, most portable
  - good for scientific programming (error, precision)
  - slowest for HPC; days your t vs computer t?
  - is the wait a problem?
  - compiler: not designed for speed
  - interpreter ⇒ *Just-In-Time* (JIT) compiler
  - class file work on all Java Virtual Machines

- **Fortran (legacy, CSE) & C (CS)**
  - compilers perfected for speed
  - examines entire code, rewrites (e.g. nested do's)
  - careful with arrays, cache lines
  - optimized for specific architectures
  - compiled code: not universal, portable
  - source code: *often* not 100% universal or portable
  - not for business, Web

Don’t be a programming bigot
“Experiment”: Good & Bad
Virtual Memory Use

- Run simple examples, look inside your computer(s)
- Measure time for each program \( \text{time in Unix} \)
- Write your own memory intensive \texttt{force} method:

\[
\begin{align*}
\text{Do } & j = 1, n \\
\text{Do } & i = 1, n \\
& f_{12}(i,j) = \text{force12}(\text{pion}(i), \text{pion}(j)) \quad \text{\( \backslash \backslash \) Fill } f_{12} \\
& f_{21}(i,j) = \text{force21}(\text{pion}(i), \text{pion}(j)) \quad \text{\( \backslash \backslash \) Fill } f_{21} \\
& f_{\text{tot}} = f_{12}(i,j) + f_{21}(i,j) \quad \text{\( \backslash \backslash \) Compute} \\
\text{EndDo} \\
\text{EndDo}
\end{align*}
\]

- Each loop iteration requires all data
- Large “working set size”
- Better \( \Rightarrow \) separate components \( \text{(next)} \)
GOOD Program, Separate Loops

- Separate loops, separate data access
  - $\approx \frac{1}{2}$ working set size
  - Groups together data used together
  - Yet, more complicated, less elegant

```plaintext
Do j = 1, n
  Do i = 1, n
    f12(i,j) = force12(pion(i), pion(j))  \ Fill f12
  EndDos

Do j = 1, n
  Do i = 1, n
    f21(i,j) = force21(pion(i), pion(j))  \ Fill f21
  EndDos

Do j = 1, n
  Do i = 1, n
    ftot = f12(i,j) + f21(i,j)  \ Compute
  EndDos
```
Try it! Java vs Fortran Optimization

- Aim: make numerically intensive program run faster
- Compare languages & their options, increasing matrix size
- Program `tune` solves matrix eigenvalue problem (power method):

\[
[H][c] = E[c] \quad (1)
\]
\[
E = ?, \quad [c] = ? \quad (2)
\]

\[
[H_{i,j}] = \begin{bmatrix}
1 & 0.3 & 0.09 & 0.027 & \ldots \\
0.3 & 2 & 0.3 & 0.9 & \ldots \\
0.09 & 0.3 & 3 & 0.3 & \ldots \\
\vdots & \vdots & \vdots & \vdots & \ddots
\end{bmatrix} \quad (3)
\]

- Almost diagonal \( \Rightarrow \ E \approx H_{i,i} \), \( c \approx \hat{c} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \end{bmatrix} \)
- \( H[2000][2000] \Rightarrow 4,000,000 \text{ elements} \)
- 11 iterations \( \Rightarrow \) smallest \( E \) to 6 places
Effects of: Dimensions, Optimize, Languages

1. Try optimize compiler options (-O1, -O2, -O3)
2. Language: ~3-10 X time difference
3. Memory: page faults

4. Loop unrolling
   - Improve memory access
   - Compiler fills cache better
   - Doubles speed (ugly)

```c
for ( i = 1; i <= L - 2; i = i + 2) {
    ovlp1 = ovlp1 + coef[i] * coef[i];
    ovlp2 = ovlp2 + coef[i+1] * coef[i+1];
    t1 = t1 + coef[j] * ham[j][i];
    t2 = t2 + coef[j] * ham[j][i+1];
}

sigma[i] = t1;
sigma[i+1] = t2;
```
High Performance Computing Lab Exercises (part II)
(examples)

Rubin H Landau
With
Sally Haerer and Scott Clark

Computational Physics for Undergraduates
BS Degree Program: Oregon State University

“Engaging People in Cyber Infrastructure”
Support by EPICS/NSF & OSU
Avoid Discontinuous Memory ⇒ page faults

- \textit{zed}, \textit{ylt}, \textit{part}: together on 1 memory page
- \textit{med2}: end of Common (after hog), different page
- Can't assure same page; improve your chances

```
Common/Double zed, ylt(9), part(9), zpart(100000), med2(9)
Do j = 1, n
  ylt(j) = zed \times part(j) / med2(9) \quad \text{// Discontinuous}
```

```
Common/Double zed, ylt(9), part(9), med2(9), zpart(100000)
Do j = 1, n
  ylt(j) = zed*part(j)/med2(9) \quad \text{// Continuous}
```

Good  \quad Bad
Important for HPC: cache misses $\Rightarrow \uparrow 10 \times T$

- **Stride**: # array elements stepped thru / op

$$\text{Tr } A = \sum_{i=1}^{N} a(i, i) \quad c(i) = x(i) + x(i + 1) \quad (1)$$

- Rule: Keep the stride low, preferably 1
  - Fortran (column): vary LH array index 1st
  - C & Java (row): vary RH array index 1st

**Method: Programming for Cache**
Exercise: Cache Misses

◆ Aim of HPC: have data to be processed in cache
  ▪ F90 2-D array storage: column-major order ↓↓↓↓
  ▪ C, Java 2-D array storage: row-major order →→→

◆ Compare times: \[ M(\text{column} \times \text{column}) \] vs \[ M(\text{row} \times \text{row}) \]

Do \( j = 1, 9999 \)
\[
x(j) = m(1,j) \quad \backslash\backslash \text{Sequential column ref}
\]

Do \( j = 1, 9999 \)
\[
x(j) = m(j,1) \quad \backslash\backslash \text{Sequential row ref}
\]

◆ 1 version always makes large jumps through memory
◆ \( \Rightarrow \) RAM: virtual memory \( \neq \) full story
Good & Bad Cache Flow
(OK to try these at home)

GOOD f90/BAD C, Minimum/Maximum Stride

Dimension Vec(idim, jdim) \ Loop A
\[
\text{Do } j = 1, \ jdim \\
\text{Do } i = 1, \ idim \\
\text{Ans} = \text{Ans} + \text{Vec}(i,j) \times \text{Vec}(i,j) \quad \text{\( \text{Stride} \ 1 \)}
\]
EndDos

Vary row rapidly

BAD f90/GOOD C, Maximum/Minimum Stride

Dimension Vec(idim, jdim) \ Loop B
\[
\text{Do } i = 1, \ idim \\
\text{Do } j = 1, \ jdim \\
\text{Ans} = \text{Ans} + \text{Vec}(i,j) \times \text{Vec}(i,j) \quad \text{\( \text{Stride} \ jdim \)}
\]
EndDos

Vary column rapidly
Exercise: Large Matrix Multiplication

\[ [C] = [A] \times [B], \quad \rightarrow \quad c_{ij} = \sum_{k=1}^{N} a_{ik} \times b_{kj} \]

- Must compromise: as both row & columns used

- Bad f90/Good C, Max/Min Stride

<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td>Do i = 1, N \ \ // Row</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Do j = 1, N \ \ // Column</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c(i,j) = 0.0 \ \ // Initial</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Do k = 1, N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c(i,j) = c(i,j) + a(i,k) * b(k,j)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EndDos</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Good f90/Bad C, Min/Max Stride

<table>
<thead>
<tr>
<th>j</th>
<th>i</th>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td>Do j = 1, N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Do i = 1, N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c(i,j) = 0.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EndDos</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Do k = 1, N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Do i = 1, N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c(i,j) = c(i,j) + a(i,k) * b(k,j)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EndDos</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>