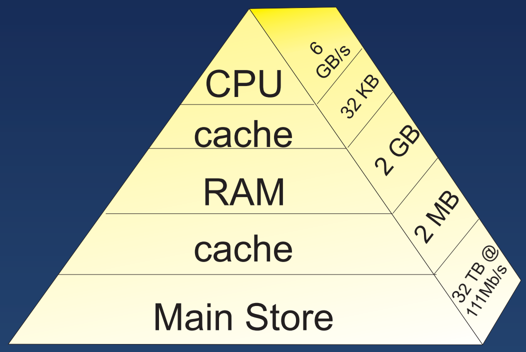
High Performance Computing Lab Exercises

(Make sense of the theory!)



Rubin H Landau

With

Sally Haerer and Scott Clark

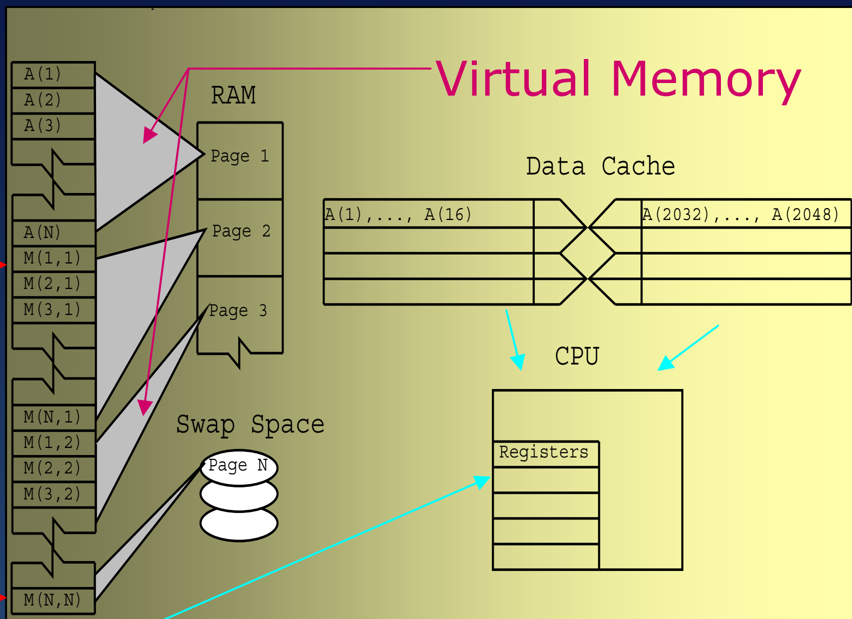
Computational Physics for Undergraduates

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 *subsets* that fit in RAM
4. Avoid simultaneous calculations on same data set [M^2/Det(M)]
5. Group together in memory data used together ($\neq \sum_i M_{ij}$)

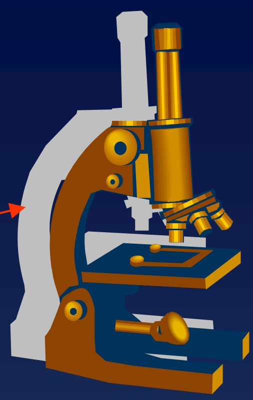
Comparison: Java *vs* F90 & C

Don't be a programming bigot

- **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**

"Experiment": Good & Bad Virtual Memory Use



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

```
Do j = 1, n
  Do i = 1, n
   f12(i,j) = force12(pion(i), pion(j))  \\ Fill f12
   f21(i,j) = force21(pion(i), pion(j))  \\ Fill f21
   ftot = f12(i,j) + f21(i,j)            \\ Compute
   EndDo
EndDo
```

  - Each loop iteration requires all data
  - Large "working set size"
  - Better $\Rightarrow$ separate components *(next)*

GOOD Program, Separate Loops

```
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
```

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

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):

$$[H]\,[\mathbf{c}] \;=\; E[\mathbf{c}] \qquad (1)$$

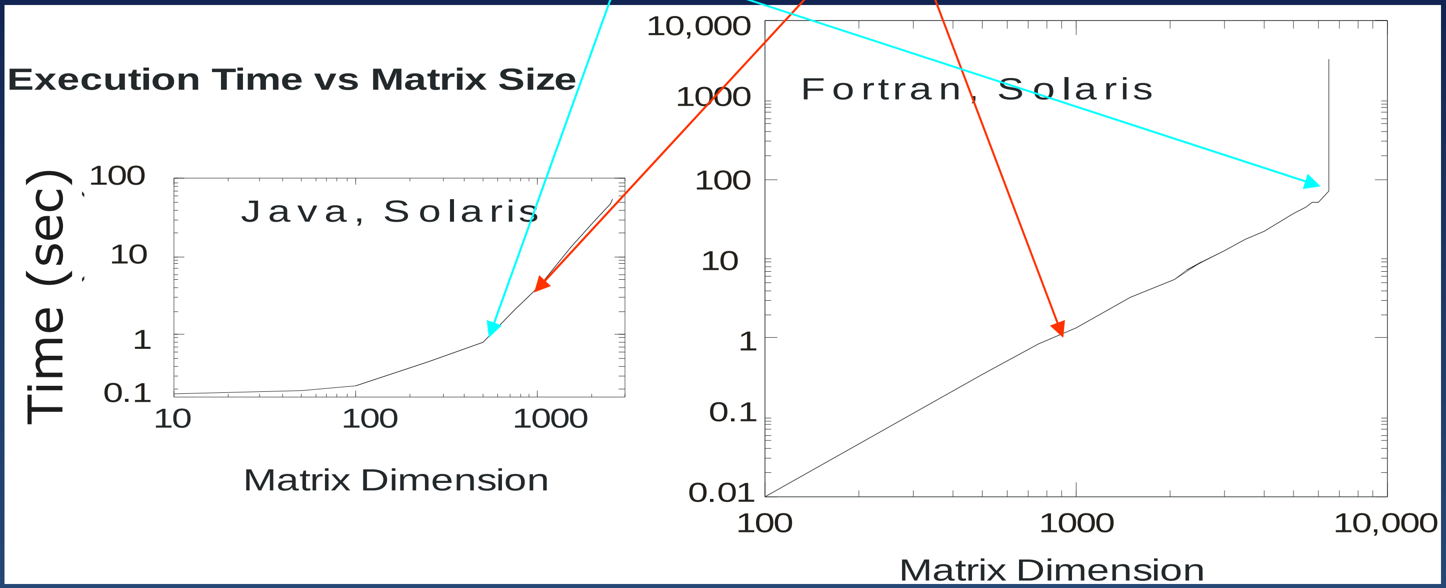
$$E = ?, \qquad [\mathbf{c}] \;=\; ? \qquad (2)$$

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

- Almost diagonal $\Rightarrow$ $E \approx H_{i,I}$, $\quad c \approx \hat{e} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \end{bmatrix}$
- `H[2000][2000]` $\Rightarrow$ 4,000,000 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

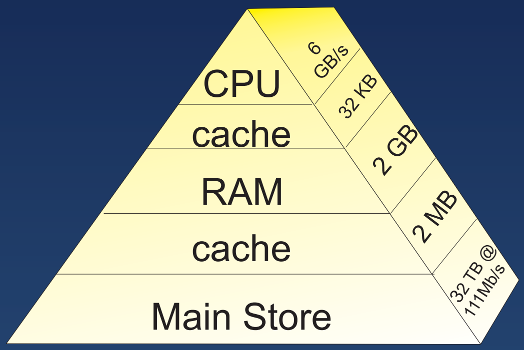


4. Loop unrolling
   - Improve memory access
   - Compiler fills cache better
   - Doubles speed (ugly)

```
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

```
Common/Double zed, ylt(9), part(9), zpart(100000), med2(9)
 Do j = 1, n
   ylt(j) = zed * part(j) / med2(9)        \\ Discontinuous
```

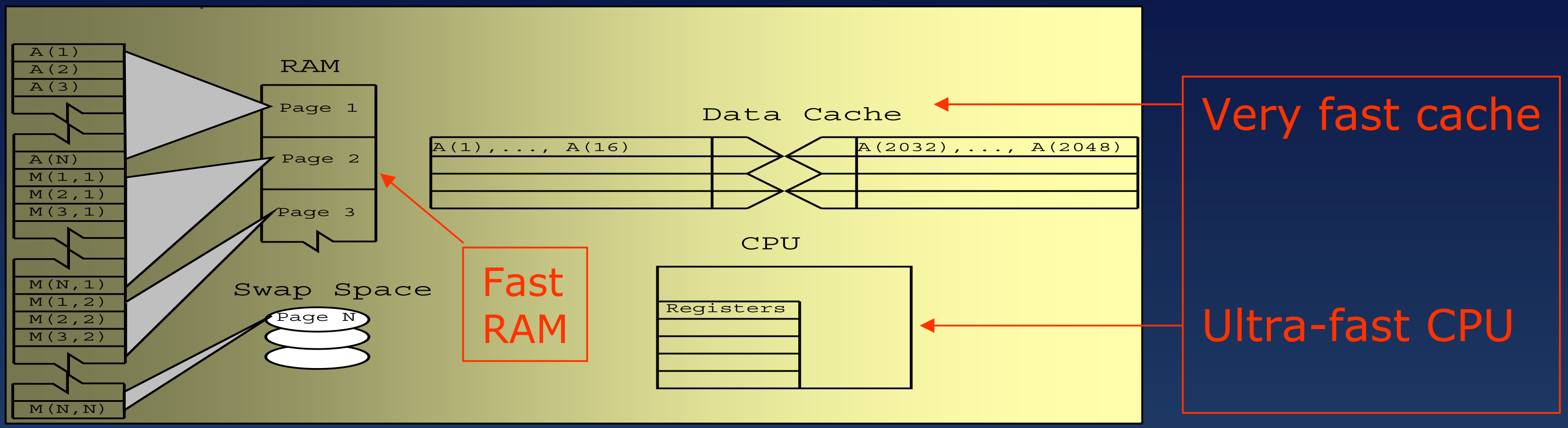
Bad

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

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

Good

Method: Programming for Cache



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

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

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

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

Exercise: Cache Misses

- Aim of HPC: have data to be processed in cache
  - F90 2-D array storage: column-major order $\downarrow\downarrow\downarrow\downarrow$
  - C, Java 2-D array storage : row-major order $\rightarrow\rightarrow\rightarrow$
- Compare times: `M(column x column)` *vs* `M(row x row)`

```
Do j = 1, 9999
   x(j) = m(1,j)          \\  Sequential column ref

 Do j = 1, 9999
   x(j) = m(j,1)           \\  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
  Do j = 1, jdim
    Do i=1, idim
    Ans = Ans + Vec(i,j)*Vec(i,j) \\ Stride 1
EndDos
```

Vary row rapidly

BAD f90 / GOOD C, Maximum / Minimum Stride

```
Dimension Vec(idim, jdim)          \\  Loop B
  Do i = 1, idim
    Do j=1, jdim
    Ans = Ans + Vec(i,j)*Vec(i,j)     \\ Stride jdim
EndDos
```

Vary column rapidly

Exercise: Large Matrix Multiplication

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

- Must compromise: as both row & columns used

- Good f90/Bad C, Min/Max Stride

```
j  Do j = 1, N
i     Do  i = 1, N
      c(i,j) = 0.0
      EndDos
k   Do k = 1, N
      Do i = 1, N
      c(i,j) = c(i,j)
               + a(i,k)*b(k,j)
      EndDos
```

- Bad f90/Good C, Max/Min Stride

```
i  Do i = 1, N        \\ Row
j     Do j = 1, N      \\ Column
      c(i,j) = 0.0     \\ Initial
k       Do k = 1, N
        c(i,j) = c(i,j)
                 + a(i,k)*b(k,j)
     EndDos
```