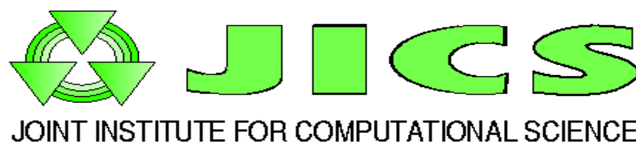
Improving Performance of Scientific Computing Codes on HPC Systems

by Christian Halloy, Ph.D.

halloy@jics.utk.edu || halloyc@ornl.gov

September 30, 2010



UNIVERSITY of TENNESSEE

OAK RIDGE NATIONAL LABORATORY
U. S. DEPARTMENT OF ENERGY

Scientific Computing? Improve?

- **Scientific Computing:**
  The use of computational resources that produce "some" science

- **Improve:**
  the end-to-end production of science has to be more efficient.
  or, in other words, *faster, bigger, more accurate, easier, ... better science!*

Ideally, the scientist should be able to concentrate on the science without worrying (too much) about the computational resources! However ….(unfortunately?) … !

- ***Improving performance of HPC codes:***
  "For a computational scientist to effectively employ parallel computing requires knowledge of parallel programming, appropriate algorithms, architecture, and software tools."
  from "A Scientist's and Engineer's Guide to Workstations and Supercomputers", 1993)

Fortunately, the scientists will have access to useful support (such as is found at NICS!)

Main Topics

- **Introduction:** Scientific Computing and Performance

- The **hardware** behind the program:
  numbers, precision, speed and space

- The **math** behind the program: Numerical Methods

- Techniques to improve **single processor** performance

- Techniques to improve **parallel programming** performance

Caveat: a) targeting scientists using HPC systems

b) not focused on specific HPC system, although some are mentioned.

Introduction: Scientific Computing and Performance

- **performance has to do with:**

==> problem size and numerical precision

==> execution time

--> turnaround of computational jobs (from submiting jobs to getting the results)

==> ease of programming

Performance ==> problem size and precision

- how big a problem can I solve?
- how precise is the numerical solution of my problem?

- **computational issues:**
  - available size of disk(s), CPU memories (RAM, caches, registers), aggregate memory (when using parallel systems).
  - machine's internal numerical representation (integers, floats/reals, double precision, etc.)
  - exact computations vs. approximations (symbolic algebra vs. floating point representations)
  - efficient data representation: clarity vs. size vs. speed

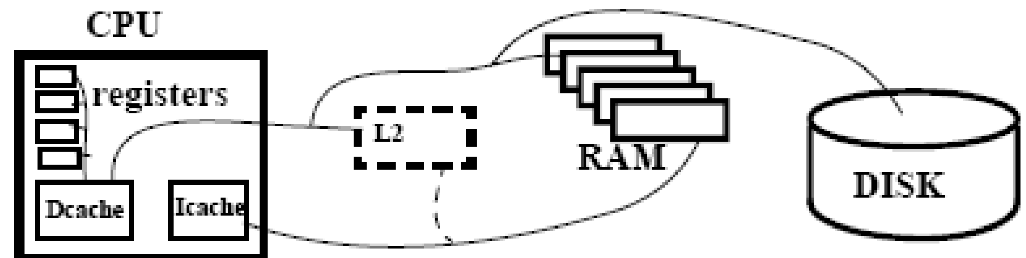
Performance ==> execution time

- how long will it take for the computer to run the program to completion?

  (but also, how long does it take for a job to start after it has been submitted to the queue!)

- **computational issues:**
  - brute speed: CPU clock (or GHz rate), and bus width

    (3 GHz does not always produce 3 GFlops! Most often it's less, … sometimes it's more!)
  - number of clock cycles per operation (logical, arithmetic, etc.);
  - multiple logical/arithmetic units on a CPU
  - access to/from disk(s), or I/O throughput (local or distributed)
  - access to/from RAM memory (local, or distributed)
  - overhead (latency), peak, and sustained bandwidth
    - I/O, RAM, CPU_to_CPU (soft/hardware latency)
  - pipelining for fetch/store of vectors or arrays
  - data cache utilization (L1, L2 and L3 cache (?))
  - instruction cache utilization
  - registers; (vector registers or units)



Performance ==> ease of programming

Once the physical problem is clearly defined and an algorithm or method is selected to solve the problem, the computational program can be implemented

computational issues:

- programming language: C, Fortran, C++, etc.
- basic requirements:
  - **program must run ! :-)**
  - **program must run "correctly" ! (validation and verification !?)**
- use debugging (and profiling) tools to help out (they're getting better!)
- suggestions for better programs:
  - code should be easy to understand;
  - include lots of "comments"
  - avoid messy "spaghetti" programming
  - make it easy to modify, and port to other computers
  - use Structured Programming
  - use subroutines, but don't overdo it
  - **use IMPLICIT NONE in Fortran!** (n1 vs. nl) or (pO vs. p0)
- time and effort of developmental stage should be "reasonable"

Example of (bad) spaghetti coding: page 1

Example of (bad) spaghetti coding: page 2

Unravelled (better) coding: page 1

```
      DO 1000 IS1 = IB1, IE1
        IN1    = ADJ(IS1)
        IF (IN1 .NE. I2) THEN

          DO 900 IS2 = IB2, IE2
            IN2    = ADJ(IS2)
            IF (IN2 .NE. I1) THEN

              J1     = IN1
              K1     = IN2
              J2     = ILIST(J1,2)
              IF (MAP(K1).GE.MAP(J2).AND.J2 .NE.I1) THEN
                K2     = ILIST(K1,2)
                IF (D(J2,K2) .NE.BIGINT) THEN
                  DIFF   = ILIST(I1,3) + ILIST(J1,3) + ILIST(K1,3) -
     &                     (C(IS1) + C(IS2) + D(J2,K2))
                  IF (DIFF.GT.SAVMAX) THEN
                    SAVMAX = DIFF
                    KGM    = 1
                    COST(KGM,1) = C(IS1)
                    COST(KGM,2) = C(IS2)
                    COST(KGM,3) = D(J2,K2)
                    GO TO 700
                  ENDIF
                ENDIF
              ENDIF
 400          J2     = IN1
              K1     = IN2
              J1     = ILIST(J2,1)
              IF (MAP(K1).GE.MAP(J2)) THEN
                K2     = ILIST(K1,2)
                IF (D(J1,K2) .NE.BIGINT) THEN
                  DIFF   = ILIST(I1,3) + ILIST(J1,3) + ILIST(K1,3) -
     &                     (C(IS1) + C(IS2) + D(J1,K2))
                  IF (DIFF.GT.SAVMAX) THEN
                    SAVMAX = DIFF
                    KGM    = 2
                    COST(KGM,1) = C(IS1)
                    COST(KGM,2) = C(IS2)
                    COST(KGM,3) = D(J1,K2)
                    GO TO 700
                  ENDIF
                ENDIF
              ENDIF
 500          J2     = IN1
              K2     = IN2
              J1     = ILIST(J2,1)
              K1     = ILIST(K2,1)
              IF (MAP(K1).GE.MAP(J2).AND.D(J1,K1).NE.BIGINT) THEN
                DIFF   = ILIST(I1,3) + ILIST(J1,3) + ILIST(K1,3) -
     &                   (C(IS1) + C(IS2) + D(J1,K1))
                IF (DIFF.GT.SAVMAX) THEN
                  SAVMAX = DIFF
                  KGM    = 3
                  COST(KGM,1) = C(IS1)
                  COST(KGM,2) = D(J1,K1)
                  COST(KGM,3) = C(IS2)
                  GO TO 700
                ENDIF
              ENDIF

 600          K1     = IN1
              J2     = IN2
              J1     = ILIST(J2,1)
              K2     = ILIST(K1,2)
```

Unravelled (better) coding: page 2

```
              IF (MAP(K1).GE.MAP(J2).AND. D(J1,K2).NE.BIGINT) THEN
                DIFF   = ILIST(I1,3) + ILIST(J1,3) + ILIST(K1,3) -
     &                   (C(IS1) + C(IS2) + D(J1,K2))
                IF (DIFF.GT.SAVMAX) THEN
                  SAVMAX = DIFF
                  KGM    = 4
                  COST(KGM,1) = C(IS1)
                  COST(KGM,2) = C(IS2)
                  COST(KGM,3) = D(J1,K2)
                ENDIF
              ENDIF
C
C           REMOVE ARCS --(I1,I2),(J1,J2),(K1,K2)-
C                        FROM LIST
 700          IF (KGM.NE.5) THEN
                IF (LISTNO.NE.0) THEN
                  DO KGM = 1, 3
                    IF (KGM .EQ. 1) THEN
                      N1     = I1
                      N2     = I2
                    ELSE IF (KGM .EQ. 2) THEN
                      N1     = J1
                      N2     = J2
                    ELSE
                      N1     = K1
                      N2     = K2
                    ENDIF
 701                LADD   = 0
                    DO L = 1,LISTNO
                      LP = L - LADD
                      IF (LIST(L,1).EQ.N1.AND.LIST(L,2).EQ.N2 .OR.
     &                    LIST(L,2).EQ.N1.AND.LIST(L,1).EQ.N2) THEN
                        LADD = 1
                      ELSE
                        LIST(LP,1) = LIST(L,1)
                        LIST(LP,2) = LIST(L,2)
                      ENDIF
 704                ENDDO
                    LISTNO = LISTNO - LADD
                  ENDDO
                ENDIF   ! of IF (LISTNO.NE.0)
C
C SUBROUTINE ARSWAP MAKES THE ARC SWAPS FOR ARCOPT
C
                IF (KGM .EQ. 1) THEN
 710              CALL ARSWAP(I1,J1,I2,K1,J2,K2,KGM)
                ELSE IF (KGM .EQ. 2) THEN
 720              CALL ARSWAP(I1,J2,K1,I2,J1,K2,KGM)
                ELSE IF (KGM .EQ. 3) THEN
 730              CALL ARSWAP(I1,J2,K1,J1,I2,K2,KGM)
                ELSE IF (KGM .EQ. 4) THEN
 740              CALL ARSWAP(I1,K1,J2,I2,J1,K2,KGM)
                ENDIF   ! of IF (KGM .EQ. 1)
 750            SAVMAX = 0
                KGM    = 5
                GO TO 100
              ENDIF   ! of  IF (KGM.NE.5)
            ENDIF   ! of IF (IN2 .NE. I1)
 900      CONTINUE   ! end of DO IS2 = IB2, IE2
        ENDIF   ! of IF (IN1 .NE. I2)
 1000 CONTINUE    ! end of DO IS1 = IB1, IE1

      IF (LISTNO.GT.0) GO TO 100
      RETURN
      END
```

?

The hardware behind the program: *precision*

Digital computers can represent only a finite set of numbers!

- **Integers are limited by the number of bits used to represent them:**

  - 2-bytes integers (16-bits): 2^{16} possibilities, usually +/- 2^{15} = 32,768

  Warning!: beware when using integer counters that might exceed this value!

  - 4-bytes integers (32-bits): 2^{32} possibilities,
    usually +/- 2^{31} = 2,147,483,648 -->(thus a 2 gig mem address limit!?)

  Often the constant MaxInt is used, and defined as MaxInt = $2^{31} - 1$
  But beware if you happen to calculate some of the following:

  Warning! ***MaxInt + 1 ; MaxInt * 2 ; MaxInt * 2.0 ;***

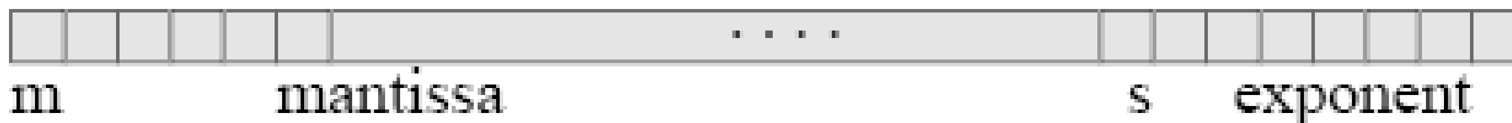
  e.g. on Cheetah: MAXINT= 2147483647 MAXINT + 1 = -2147483648
  Kraken, etc: MAXINT * 2 = -2 MAXINT * 2.0 = 4.294967296E+09

  Results may vary w/hardware, or compilers. Check your system's "MaxInt" value, to be sure

Internal representation: floating-point numbers

- **arithmetic Real numbers are approximated with *floating-point numbers***

- 32-bits (4 bytes) floats (or reals) use ***24 bits for the mantissa*** (including sign) and ***8 bits for the exponent*** (including sign)



$$x = (-1)^m \text{ mantissa} . 2^{(s).\text{exponent}}$$

$$\text{mantissa} = b_1.2^{-1} + b_2.2^{-2} + b_3.2^{-3} + ... + b_{23}.2^{-23}$$

Warning: *computers read/write floats in different "endian" ways!*

- usually the mantissa is chosen such that b1=1, but then b1 (the phantom bit) can be dropped and we get an extra bit of precision!
- but still only 2^{24} = **16,777,216 possible mantissa values!**

floating-point numbers (cont'd)

- **limitations: (single precision, 32-bits)**

  - the **biggest** number before "overflow" is
  $\mathbf{2}^{(s).\text{exponent}} = \mathbf{2}^{+127} = \mathbf{1.7\ 10}^{+38}$

  - the **smallest** number before "underflow" (or zero) is
  $= \mathbf{2}^{-127} = \mathbf{5.9\ 10}^{-39}$ (but often $\mathbf{2}^{-149} = \mathbf{1.4\ 10}^{-45}$)

  - the **machine precision** (epsilon) is defined as the smallest value of epsilon such that:
  1.0 + eps > 1.0 still holds true; [**100000000000000000000001]**
  $\mathbf{eps} = \mathbf{2}^{-23} = \mathbf{1.2\ 10}^{-07}$ (sometimes $\mathbf{eps} = \mathbf{2}^{-24} = \mathbf{6.0\ 10}^{-08}$)

- Note: only rational numbers of the form p/q can be represented exactly!
where p = an integer in the range +/- $\mathbf{2}^{23}$ and q = is a power of 2
(thus, 1/10, 1/3, 5/6, 2/7 are only approximated!)

floating-point numbers (cont'd)

- **limitations: (double precision, 64-bits)**

  - 64-bit floats/reals use ***53 bits for the mantissa*** (including sign) and ***11 bits for the exponent*** (including sign)

  - the biggest number before "overflow" is
  $\mathbf{2}^{(s).\text{exponent}} = \mathbf{2^{+1023} = 9.0\ 10^{+307}}$

  - the smallest number before "underflow" (or zero) is
  $= \mathbf{2^{-1023} = 1.1\ 10^{-308}}$ (sometimes $\mathbf{2^{-1074} = 4.9\ 10^{-324}}$)

  - the machine precision (epsilon) for which
  1.0 + eps > 1.0 still holds true is:
  $\mathbf{eps = 2^{-52} = 2.2\ 10^{-16}}$ (sometimes $\mathbf{eps = 2^{-53} = 1.1\ 10^{-16}}$)

Precision problems: simple examples

Square root function:

sqrt(2.0) * sqrt(2.0) - 2.0 = -1.19209×10^{-07} on Kraken (-0.684570E-07 on Cheetah)

sqrt(1000.0) * sqrt(1000.0) - 1000.0 = -6.10352×10^{-05} Kraken (-0.3606196E-04 Cheetah)

however!: sqrt(10000.0) * sqrt(10000.0) - 10000.0 = 0.0

you may get different results on different computers!

Combining BIG numbers and a small number: (with a = 1.0/3.0)

[for example: *1,000,000* + *0.333333*]

value of b ==>	$1.0\times10^{+07}$	$5.0\times10^{+06}$	$1.0\times10^{+06}$	$1.0\times10^{+05}$
b - b + a =	0.333333	0.333333	0.333333	0.333333
b + a - b =	0.0	0.5	0.3125	0.335938
a + b - b =	0.0	0.5	0.3125	0.335938
a - b + b =	0.0	0.5	0.3125	0.335938

Precision problems: simple examples (cont'd)

Solving $x^2 - 1000 x + 1 = 0$

[check how good solution is by recalculating: **$f(x) = a x^2 + b x + c$**; is *$f(x) => 0$*?]

direct solve: $x_{1,2} = (-b +/- sqrt(b^2 - 4 a c))/(2 a)$

$x_1 = 1.00708*10^{-03}$ $x_2 = 999.999$

$f(x_1) = -7.08*10^{-03}$ $f(x_2) = 6.25*10^{-02}$ (on Kraken)

successive approximations:

$x_1 = - (c + a x_1^2) / b$ **$x_2 = - (b + c / x_2) / a$**

1) $x_1 = 1.0*10^{-03}$ $x_2 = 1000.0$

2) $x_1 = 1.000001*10^{-03}$ $x_2 = 999.999$

3) $x_1 = 1.000001000002*10^{-03}$ $x_2 = 999.9989999990$

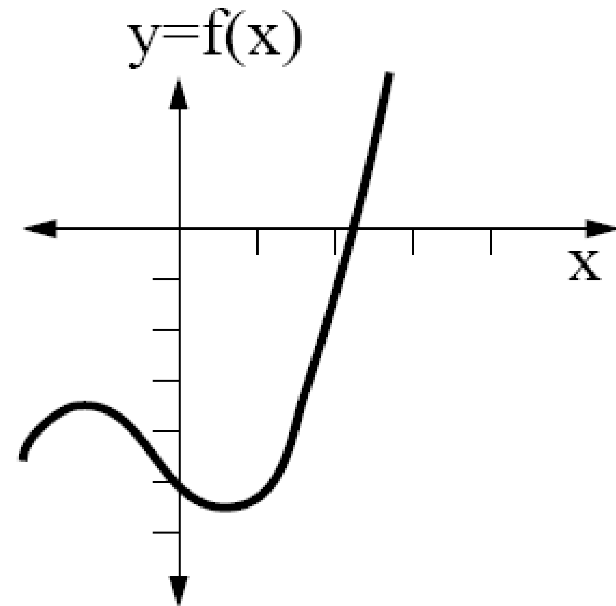
$f(x_1) = < 1.0*10^{-16}$ $f(x_2) = 1.16*10^{-10}$

Precision problems: simple examples (cont'd)

Solving $x^3 - 2x - 5 = 0$; solution between 2.0 and 3.0

rewrite equation with $x = 2 + z$
obtain: $z^3 + 6z^2 + 10z - 1 = 0$
iterate on $z = (1 - 6z^2 - z^3)/10$

$z = 0.1$	$f(x) = 6.1*10^{-02}$
$z = 0.0939$	$f(x) = -7.3*10^{-03}$
$z = 0.09462688$	$f(x) = 8.4*10^{-04}$
$z = 0.09454272$	$f(x) = -9.8*10^{-05}$
$z = 0.09455250$	$f(x) = 1.1*10^{-05}$
$z = 0.09455136$	$f(x) = 1.3*10^{-06}$



after the 12th iteration we obtain (on Kraken compute nodes):

$z = 0.094551481542037$; $f(x) = -3.2*10^{-12}$

Precision problems: simple examples (cont'd)

Forward and Backward Summations

$$\sum_{n=1}^{M}\left(\frac{1}{n}\right) = 1+\frac{1}{2}+\frac{1}{3}+\frac{1}{4}+\ldots+\frac{1}{M-1}+\frac{1}{M} \Rightarrow \log(M)_e + Gamma$$

this sum diverges when $M \to \infty$; Gamma = 0.5772156649 (Euler's constant)

	forward sum	backward sum	ln(M)+Gamma
M = 1.0e06	14.3574	14.3927	14.3927
M = 1.0e07	15.4037	16.6860	16.695311
M = 1.0e08	15.4037	18.8079	18.997896
M = 1.0e09	15.4037	18.8079	21.300482
M = 2.147e09	15.4037	18.8079	22.064778
M = 1.0e07	16.695311531453	16.695311531453	doubleprecis
M = 1.0e08	18.997896442992	18.997896442992	doubleprecis
M = 1.0e09	21.300481541265	21.300481567974	doubleprecis
M = 2.147e09	22.064778332636	22.064778327573	doubleprecis

Solving A x = b : *Beware of ill-conditioned matrices!*

Suppose we have the following 3x3 ill conditioned matrix:

	-73.0000	78.0000	24.0000
A =	92.0000	66.0000	25.0000
	-80.0000	37.0000	10.00000

Calculting the determinant of A gives:

det (A) = D = 1.00000

Now, modify only the element A(3,3) by Epsilon = 1.00000E-02

A(3,3) = A(3,3) + Epsilon = 10.01000

Recalculate the determinant of this new A matrix:

det (A+Epsilon) = -118.945 !!!

Precision problems: simple examples (cont'd)

Problem: *is cos(n*360 + 60) = 0.5 ?   with n = 1, 2, 3, …*

Expansion series for cos(x):

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \ldots + (-1)^n \left(\frac{(x)^{2n}}{(2n)!} \right); n = 0,1,2,3,\ldots$$

first method:

```
pi = 3.1415926536 ; angle = 60.0 ; precision = 1.0e-06 ; term = 1.0
x = angle * pi / 180.0 ; mycos = 1.0 ; p = 0 ; factp = 1 ; sign = 1
  dowhile (term .gt. precision)
    sign = -sign
    p = p + 2
    factp = factp * (p-1)*p
    term = x^p / factp
    mycos = mycos + term * sign
  end do
```

WARNING! This program OVERFLOWS with x = 10 or greater!

Precision problems: the cos series (cont'd)

So, what's the problem? Remember that:

$10! = 3.6*10^{+06}$; $20! = 2.4*10^{+18}$; $30! = 2.6*10^{+32}$; $\mathbf{40! = 8.2*10^{+47}}$
$50! = 3.0*10^{+64}$; $60! = 8.3*10^{+81}$; $\mathbf{70! = 1.2*10^{+100}}$; $80! = 7.2*10^{+118}$

So, if x =10, we need to include about n=20 terms of the series, such that:
$2n = p = 40$, and **term** = $\mathbf{x^p}$ **/ factp** = $10^{+40} / 40!$ **< precision =** $\mathbf{1.0*10^{-06}}$

second method:

avoid the overflow of the denominator "factp" by calculating each **"term"** from the previous one. The quotient will not grow so quickly.

new_term = previous_term * x * x / (p * (p - 1))

"term" will grow until **p > x**, it then decreases progressively from there on.

Precision problems: the cos series (cont'd)

PRGRAM COS(X) 2:

```
pi = 3.1415926536 ;   angle = 60.0 ;   precision = 1.0e-06 ;   term = 1.0
x = angle * pi / 180.0 ;   mycos =   1.0 ;   p = 0 ;   factp = 1 ;   sign = 1
  dowhile (term .gt. precision)
    sign = -sign
    p = p + 2
    term = term * x * x / ((p-1)*p)         (or use x2 = x * x)
    mycos = mycos + term * sign
  end do
```

WARNING!

method 2 gives COS(X) > 1 for X = 20 or greater ! (single precision)

(or, for X = 40, in double precision)

CAUSE:

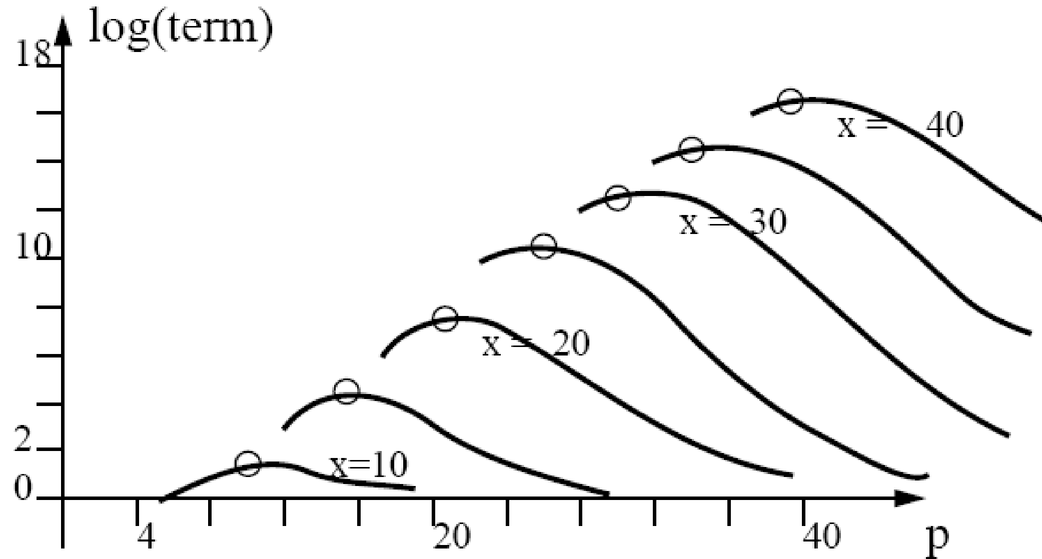
We are still adding and substracting VERY big terms together!

(see graph on next slide)

Precision problems: the cos series (cont'd)

term = x^p / p!

the maximum term is found at p=x (approximately)



[e.g: for x=30 you could have 12345678901.23 - 12345660000.23 = 18901.00]

======= ======= =

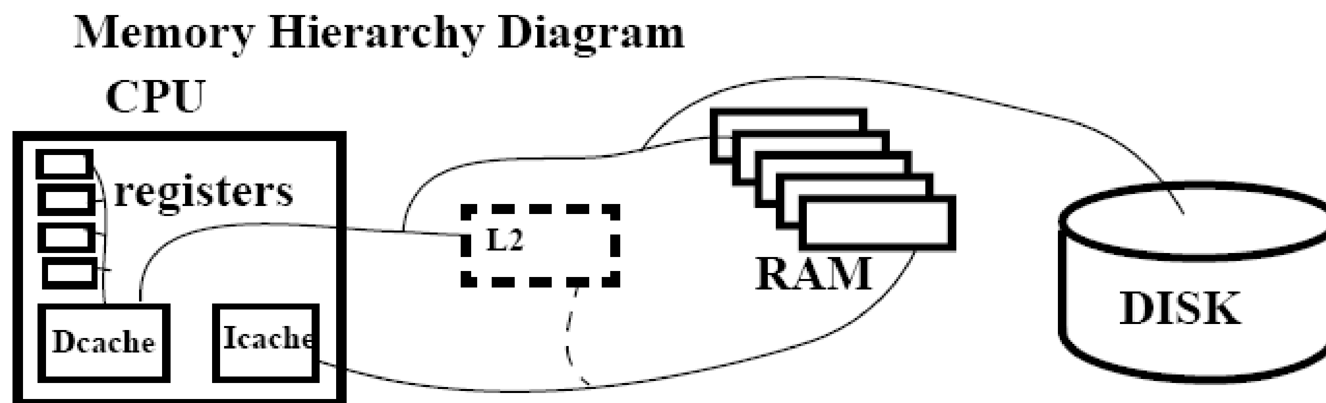
Finally... the BEST SOLUTION: ***first reduce the angle "x" to the first quadrant***

0 <= x <= pi/2 (of course! :-)

For other series (Bessel, Legendre, …) check out Abramovitz & Stegum's "Handbook of Mathematical Functions"

Single processor performance: Memory Management

a lot of time is spent accessing/storing data from/to memory. It is important to keep in mind the relative times for each memory type:



Approximate access times:
-CPU-registers: 0 cycles (that's where the work is done!)
-L1 Cache: 1 cycle (Data & Instruction cache)
-L2 Cache (static RAM): 3-5 cycles?
-Memory (DRAM): 8-12 cycles (Cache miss); 30-60 cycles TLB update
-Disk: about 100,000 cycles!

(-connecting to other nodes: 10,000 cycles and more depending on network latency)

Memory Management (cont'd)

Things to keep in mind:

- **Registers:**
  - computations benefit from re-using data stored in registers (loops or iterations working on few variables, constants, …)

- **Data Cache:**
  - if data fits in cache , computations are much more efficient. For example, on Kraken, L1 = 64 KB (14.5 GB/s), L2 = 512 KB (~8.5 GB/s), L3 = 8 MB (? GB/s), main Mem 16 GB (3 GB/s).
  - Repeated access to an L1 cache takes only 1 cycle

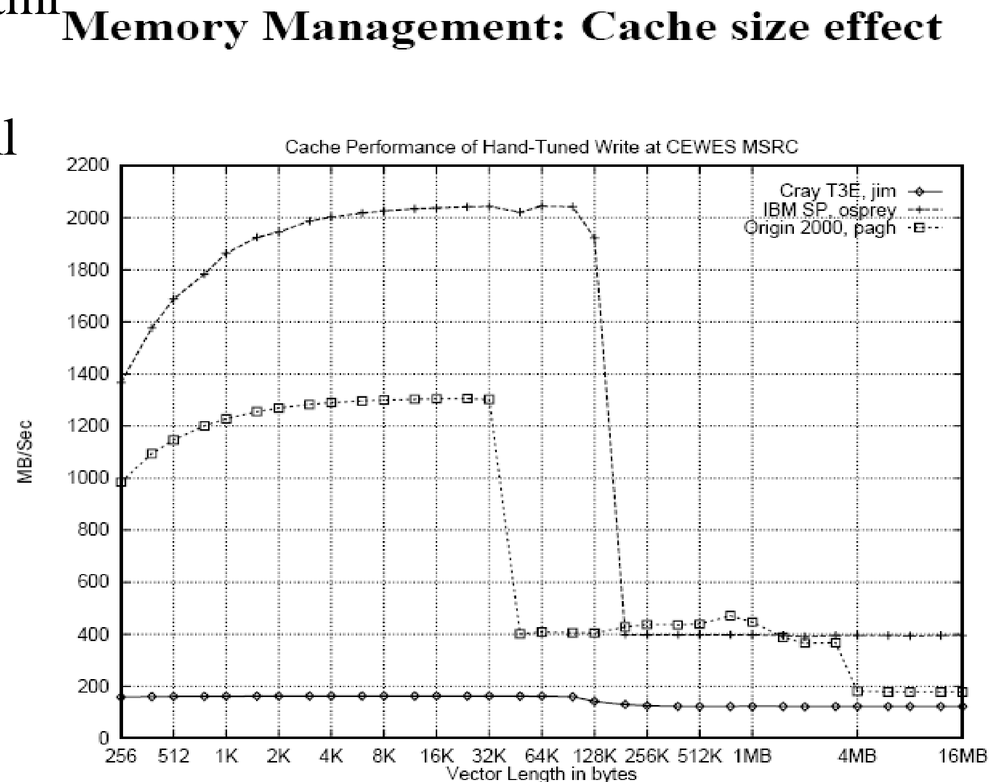
- **Instruction Cache:**
  - similar in speed to the Data Cache, they are used only to store instructions. Useful with loops, specially when the whole loop fits within the Instruction Cache

- **Main memory (DRAM):**
  - if possible, do computations on data that fit, not only within the DRAM size, but if possible within the size of memory handled by the ***Translation Lookaside Buffer*** (TLB)
  - crossing (4K?) page boundary costs an extra ~30 to 60 cycles
  - switching from read to write costs and extra ~6 cycles
  - a page fault (data not in main memory, need to go to disk) costs ~1 milisec (~1,000,000 cycles)!

Memory Management (cont'd)

- **Finding out about your current system:**
  - search in the hardware manuals (do they exist!?)
  - ask your local sys-admin (!?) or email help@teragrid.org ! ☺
  - do some experimenting on our own:
- **Also:** take a look at http://icl.cs.utk.edu/projects/llcbench/
  and within that page: /cachebench.html
  /mpbench.html
  /blasbench.html

Examining a cachebench graph:



Memory Management (cont'd)

Arrays are allocated differently within memory when programming in C or Fortran!

- data arrays are (usually) stored contiguously in memory
- working on data stored contiguously in memory improves performance due to the pipelining of data reads/writes

Stride minimization

- the stride is the distance (in bytes) between successive data elements required by a loop. A stride of 1 is optimal and it is also the default value when going along columns (in Fortran) or along rows (in C)
  MINIMIZE THE STRIDE TO MAKE BETTER USE OF THE DATA CACHE!

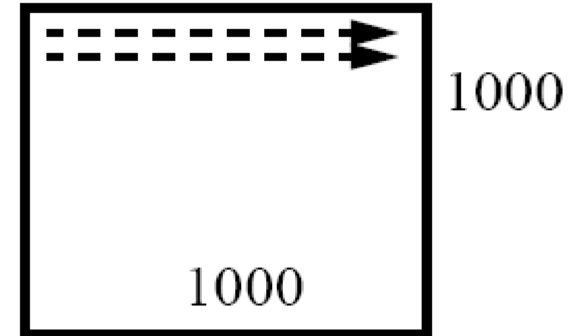
THUS: in **C (row major order)** make sure the ***rightmost subscript (or array index)*** is modified in the ***innermost loop*** of the code

: in **Fortran (column major order)** make sure the ***leftmost subscript (or array index)*** of is modified in the ***innermost loop*** of the code

Memory Management: examples in C

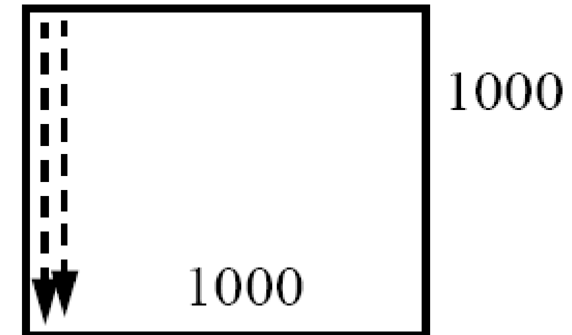
Example: C program with stride-1

```
for(i=0 ; i<1000 ; i++)
   for(j=0 ; j<1000 ; j++)
      c[i][j] = c[i][j] + a[i][j] * b[i][j]  ;
```



Example: C program with stride-1000

```
for(j=0;j<1000;j++)
   for(i=0 ; i<1000 ; i++)
      c[i][j] = c[i][j] + a[i][j] * b[i][j]  ;
```



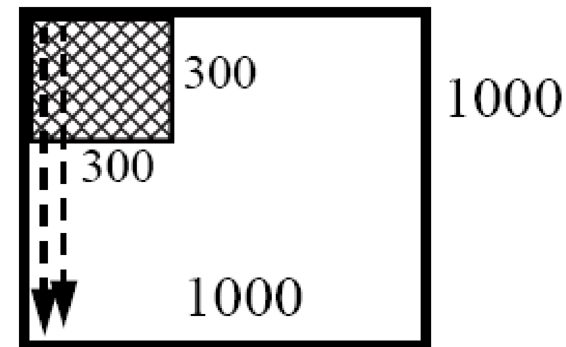
The stride-1 loop runs almost 40 times faster than the stride-1000 loop!
(with –O3 optimization flag).

Single processor performance: Additional Hints

- once the program is debugged, **declare arrays to exact sizes** whenever possible. This reduces memory use and also optimizes pipelining and cache utilization

i.e. *don't declare A(1000,1000) if you only need a matrix of size A(300,300)*

drawback: this reduces the flexibility of the program.



- maybe keep a separate "fine-tuned" version of the program for production runs. Use the other one for ease-of-programming and to check results.

- Eliminate unecessary I/O (reads and writes to disk are **expensive**!)

- Use unformatted (binary) I/O whenever possible, to save time AND disk space! e.g Storing numbers such as_**-123.456789012345_E+023_** in text (ASCII) format takes up 24 bytes for each number, while a double precision numbers saved in binary format would takes up only 8 bytes each.

Improving your code: Loop Optimizations

many scientific programs spend a great amount of time within loops. Thus, **loops must be optimized!** Some of the techniques used are:

- Stride minimization (seen already)
- Loop Fusion
- Loop Interchange
- Invariant Code Extraction
- Loop defactorization
- Neighbor Data Dependency
- Multi-nested loops collapse
- Loop Unrolling
- Loop Unrolling and Sum Reductions
- Outer Loop Unrolling
- IF loops, WHILE loops, and DO loops

Topics for a whole separate lecture! (or workshop!?)

Improving single processor performance with compiler switches

Compiler Optimization Switches

Fortran and C compilers have different levels of optimization that can do a fairly good job at improving a program's performance. The level is specified at compilation time with a **-O** switch (see "**man f77, or pgf90, or ftn**" and "**man cc, or pgcc**" on your machine!)

- a same level of optimization on different machines will not always produce the same improvements (don't be surprised!)

-O is either *no optimization*, or *maximum optimization* !

-O2 (same as -O on some machines) simple inline optimizations

-O3 (and **-O4** on some machines) more complex optimizations designed to pipeline code, but may alter semantics of program

IBM SP specific: - combine -O3/4 **-qstrict** to preserve semantics, **-**qtune=pwr4, -qhot

Cray X1E specific: -h ssp, -hlist=m or –rm (loopmark listings), -h aggress

Cray XT3/4/5 specific: -fastsse, -Mcache_align, -Munroll=c:4, -tp k8-64 (Opteron), -Mipa=fast, -small_pages, etc, etc.

- WARNING: debugging (**-g** switch) usually cancels optimization!

More info at http://cray.docs.com

(and of course, http://info.nccs.gov/resources/<machinename>

Improving single processor performance: food for thought

- Yet, it is *easy* to write inefficient codes that run at only 1% (or even less!) of peak performance on a single processor

  Parallelizing an inefficient sequential code is a waste of time !

- It is the programmer's responsibility to take advantage as much as possible of the CPU's hardware and software characteristics to boost the performance of the program! Quite often, just a few simple changes to one's code improves performance by a factor of 2, 3, or even better! However, don't overdo this if the code is to be portable!

- Also, simply compiling with some well chosen optimization flags (-O3, -qtune, -fastsse, ...) can improve performance dramatically!

Useful Reminder:

always add a comment at the top of your code indicating how you compiled it!

e.g.:

Compiled with: f77 -O3 -qtune=pwr4 -qstrict -o mycode mycode.f -lm -lmpi -lblas

/*compiled with: xlc_r -O3 -qtune=pwr4 -qstrict -o mycode mycode.c -lm -lpessl */

or, use makefiles but name them accordingly: *make.mycode, make.tunedcode,*

Back to: the math behind the program

Choosing the appropriate algorithm is crucial!

The most elegant algorithm might not be the most efficient. Consider also that a different method might be more appropriate for a parallel computer.

- **modifying the numerical algorithm can make you gain a significant amount of speed and/or memory**

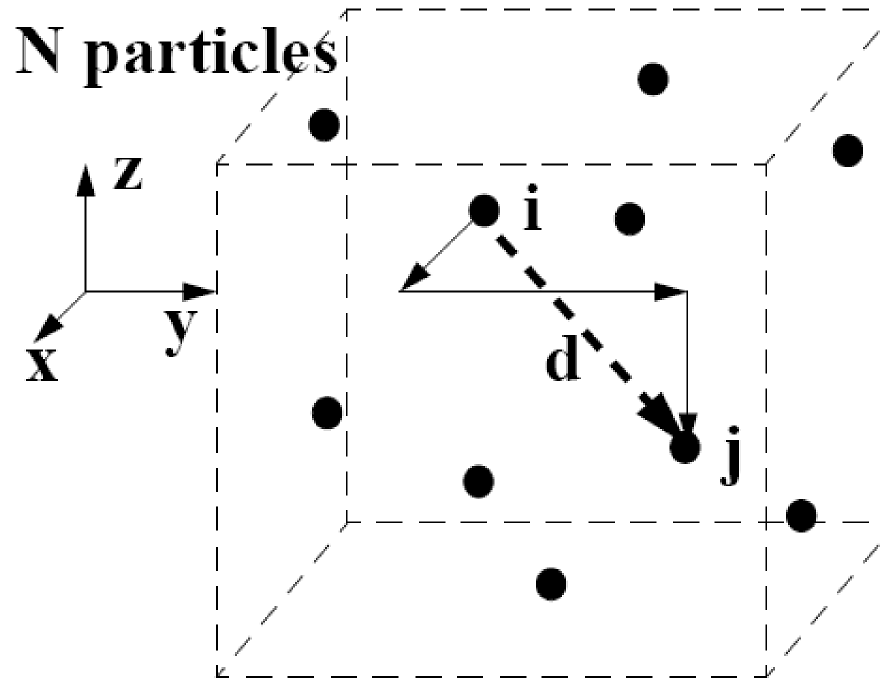
  - no point in optimizing a code if the numerical method is lousy from the beginning! (e.g. Cramer's rule for determinants in LinAlg)

- **use math-library subroutines (with caution! Verify utilization and output)**

  - ***blas, scalapack, essl, libsci, scsl, fftw, etc***. found on each system
  - see also: Numerical Recipes in C/Fortran/F90, and other books, but be aware that these subroutines might not always be well suited for parallel computing
  - whenever possible, use parallel versions: pblas, blacs, scalapack, parpack, pessl, petsc, aztec, etc., and/or the vendor's scientific libs!
  - They are usually fine-tuned to the specific hardware as well as to the sizes of the array variables that are sent to them; (block-parameters)
  - Occasionally additional improvement can be obtained by "cleaning up" the math routines to fit the specific needs of your problem. This will reduce the overhead associated with all the special cases that these routines consider.

Math optimizations: Nearest Neighbor Problem

Many physical problems require calculations of forces or potentials between neighboring particles (molecules, atoms, stars, etc.)



Problem: find which is the closest neighbor to each particle, based on the well known euclidean distance:

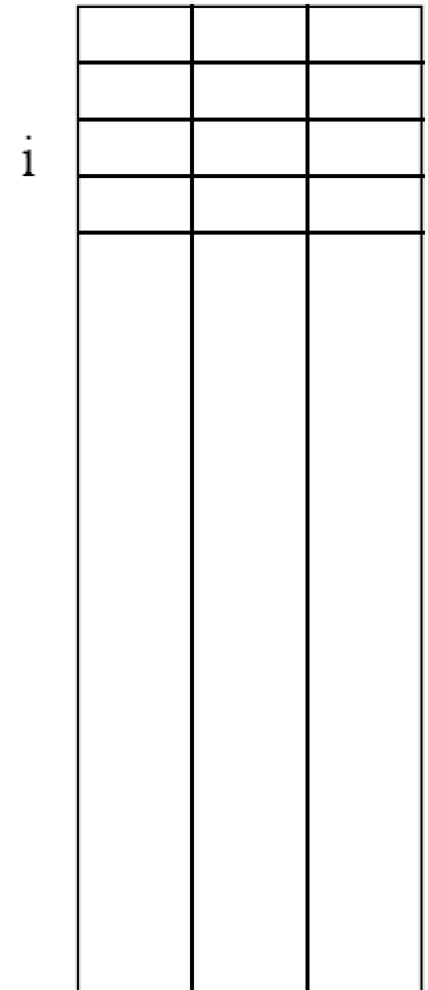
$$\mathbf{d_{ij} = sqrt\ [\ (\ x(i) - x(j)\ )^2 + (\ y(i) - y(j)\ )^2 + (\ z(i) - z(j)\ )^2\ ]}$$

Nearest Neighbors: 1st (intuitive) Solution

```
/* initialize randomly the positions of all particles */
 for (i = 0 ; i < NUMPART ; i++)
   { x[i] = drand48() * 20 - 10; y[i] = drand48() * 20 - 10;
     z[i] = drand48() * 20 - 10; }
   gettimeofday(&t1,NULL);
 for (i = 0 ; i < NUMPART ; i++) {
    min = 60.0;
    for (j = 0;j < NUMPART;j++) {
      dist = sqrt( ((x[i] - x[j]) * (x[i] - x[j])) + ((y[i] - y[j]) * (y[i] - y[j])) +
                   ((z[i] - z[j]) * (z[i] - z[j])));
      if ((dist < min) && (i != j)) {
         min = dist;
         nn[i] = j;
       }
    }
 }
 gettimeofday(&t2,NULL);
```

Nearest Neighbors: 1st (modified) Solution

```
for (i = 0;i < NUMPART;i++){
  xyz[i][0] = drand48() * 20 - 10; /* x(i) = xyz [i][0] */
  xyz [i][1] = drand48() * 20 - 10; /* y(i) = xyz [i][1] */
  xyz [i][2] = drand48() * 20 - 10; /* z(i) = xyz [i][2] */
}
gettimeofday(&t1,NULL);
for (i = 0;i < NUMPART;i++){
  min = 60.0;
  for (j = 0;j < NUMPART;j++){
    dist = sqrt( (( xyz[i][0] - xyz[j][0]) * ( xyz[i][0] - xyz[j][0])) +
                 (( xyz[i][1] - xyz[j][1]) * ( xyz[i][1] - xyz[j][1])) +
                 (( xyz[i][2] - xyz[j][2]) * ( xyz[i][2] - xyz[j][2])));
    if ((dist < min) && (i != j)){
      min = dist;
      nn[i] = j;
     }
   }
 }
gettimeofday(&t2,NULL);
```



Nearest Neighbors: 2nd Solution

Observation:

if dij is the minimum distance between particle "i" and all the rest of the particles, then dij^2 is also a minimum

(Math justification: the sqrt and ^2 functions are smooth monotonous, ... etc.)

Conclusion:

get rid of the sqrt() function !

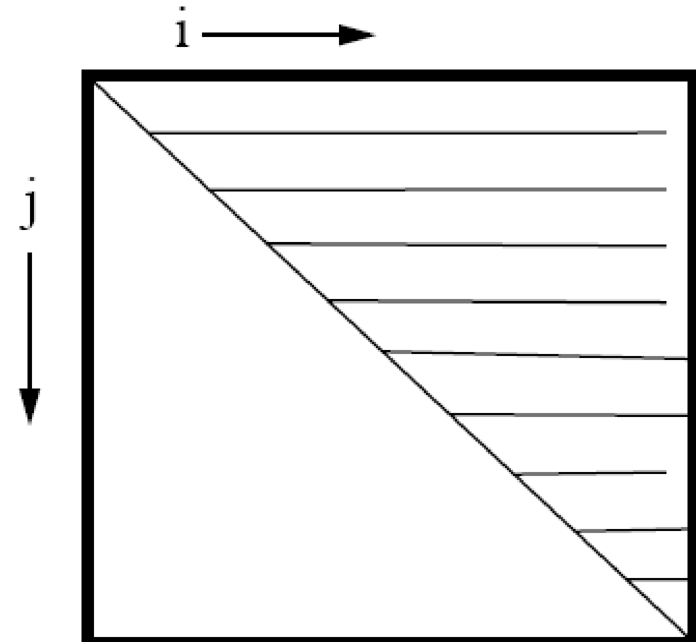
Result:

code will run almost twice faster !

Nearest Neighbors: 3rd Solution

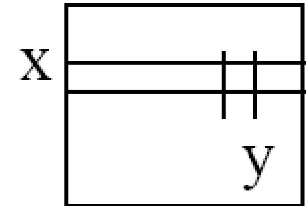
Method: Instead of doing **N^2** distance comparisons, do only **Nx(N-1)/2**
Result: code will increase speed by ***another factor of 2 !***

```
for (i=0; i<(NUMPART-1); i++){
  for (j = (i+1);j < NUMPART;j++){
    dist = (( xyz[i][0] - xyz[j][0]) * ( xyz[i][0] - xyz[j][0]))+
           (( xyz[i][1] - xyz[j][1]) * ( xyz[i][1] - xyz[j][1]))+
           (( xyz[i][2] - xyz[j][2]) * ( xyz[i][2] - xyz[j][2]));
    if ((dist < xyz[i][3])){
    xyz[i][3] = dist;
    nn[i] = j;
    }
     if ((dist < xyz[j][3])){
    xyz[j][3] = dist;
    nn[j] = i;
    }
  }
}
```

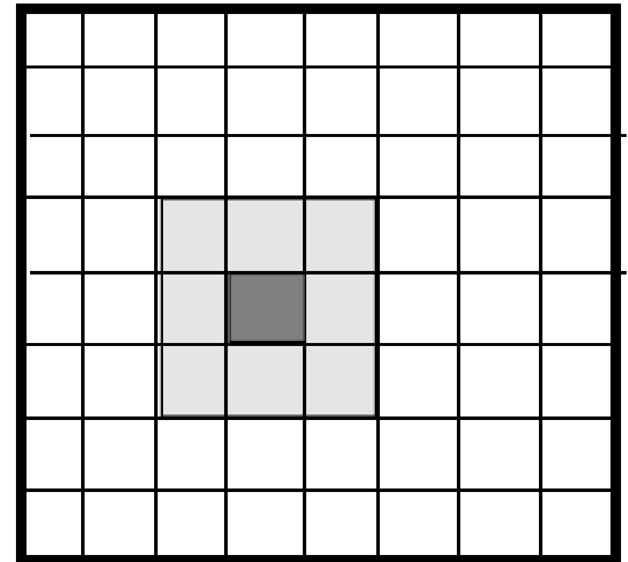


Nearest Neighbors: Other Methods

Assuming particles interact only within a certain sphere of radius = “R”
- scan first for x(i)-x(j) < R
  - then test for y(i)-y(j) < R
    - finally test for z(i)-z(j) < R
      - only then, calculate “dist” and keep track of minimum.

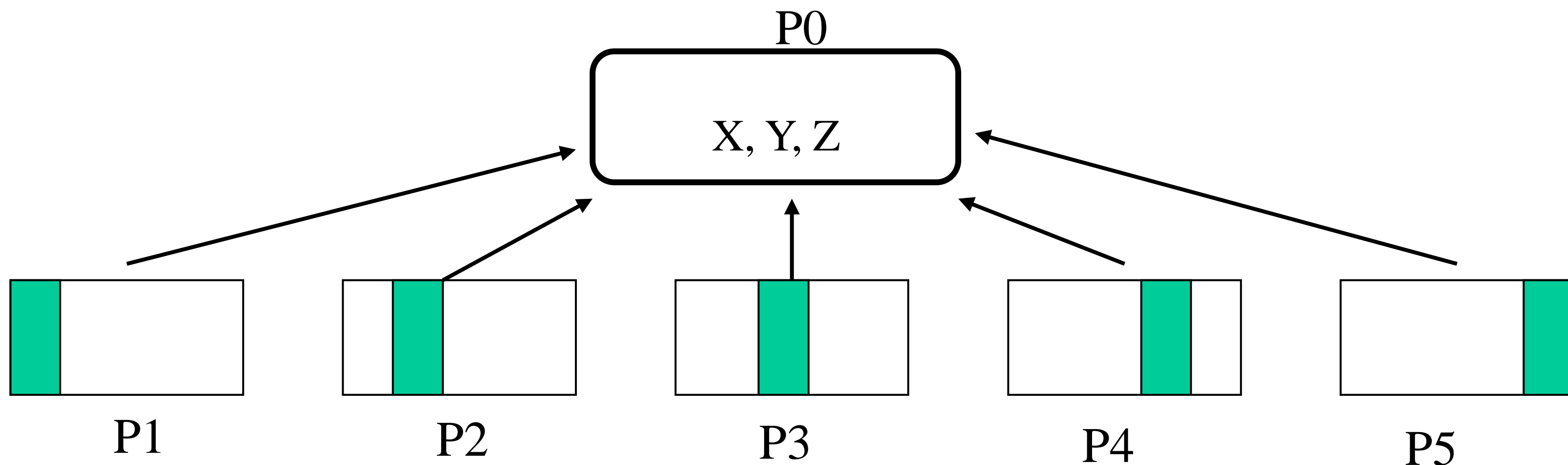


Use a Particle-in-Cell (PIC) method:
- space is divided into a grid of cells
- size of each cell is < R (or = R)
- particles are ordered according to physical cells coordinates.
- distances are calculated within cells
  - and then with 8 neighboring cells (or 26, in 3-Dimensional case)



Nearest Neighbor Example with Message Passing Method 1

- With N processors available, subdivide the work into N equal tasks.
  - The master generates randomly the X,Y,Z coordinates of 5000 particles
  - The master sends a copy of all the X,Y,Z coordinates to each processor (3 x 5000 x 8 bytes = ~120KB) -this is sequential and repeated 5 times!
  - Each processor calculates a subset of 5000/N minimum distances
  - Each processor sends back its subset list of nearest neighbors
  - the master node receives and concatenates the information



N N Example with Message Passing Method 2

a) master generates randomly the X,Y,Z coordinates of 5000 particles

b) sends only a subset 5000/N of the X,Y,Z coords to each processor

c) each processor keeps a local copy of its own subset of coordinates

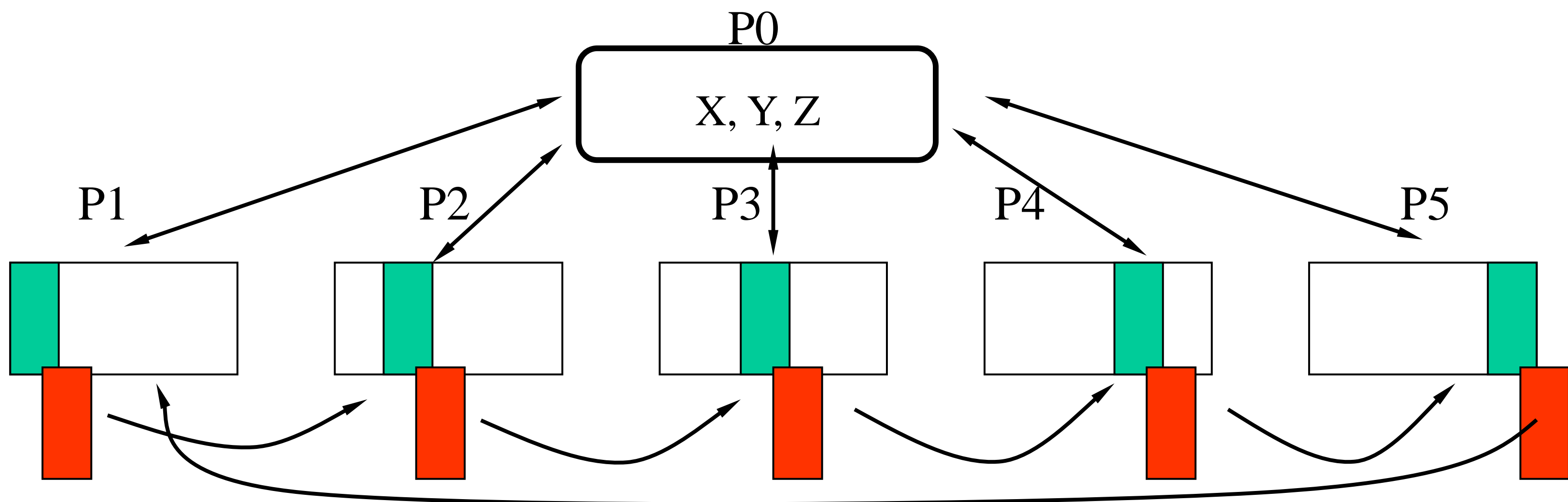
d) each processor calculate distances and temporary minimums between its own subset and the current replica of 5000/N coordinates

e) each processor sends to the next processor in round-robin fashion the replica of 5000/N coordinates.

f) repeat steps d) and e) N times

g) Each processor sends back its subset list of nearest neighbors

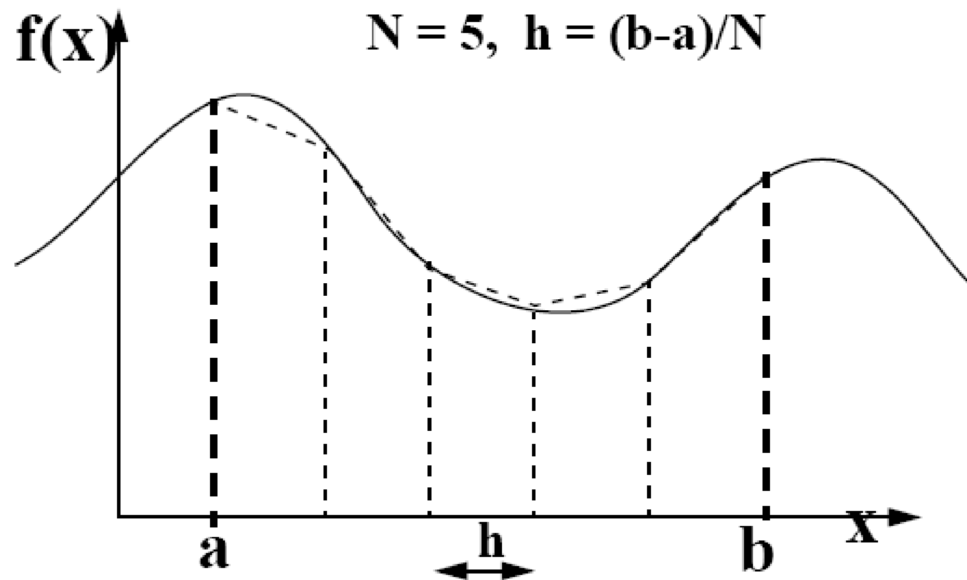
h) the master node receives and concatenates the information



Numerical methods example: calculating integrals - trapezoidal rule

Problem:

$$I = \int_a^b f(X)\, dX$$



1st order method: or *trapezoidal rule*

- subdivide **(b-a)** into **N** intervals of width **h = (b-a)/N** and do a linear interpolation between adjacent $f(x_i)$

Solution: $I = h * (f_0/2 + f_1 + f_2 + ... + f_{N-1} + f_N/2) + O(h^2)$
with $f_i = f(x_i)$ and $x_i = a + i * h$, $i = 0, 1, ..., N$

calculating integrals: Simpson's rule

2nd order method: or *Simpson's rule*

- subdivide **(b-a)** in **N** intervals and do a cubic spline interpolation between every 3 adjacent **$f(x_i)$** (parabolic fitting)

Solution: (N needs to be even)

$$I = h/3 * (f_0 + 4*f_1 + 2*f_2 + 4*f_3 + ... + 2*f_{N-2} + 4*f_{N-1} + f_N) + O(h^4)$$

comparing both methods:

if $h = 1/1000 = 10^{-3}$, then for a similar amount of computations the errors will be *proportional* to O() 10^{-6} and 10^{-12} respectively !

3rd and higher order methods:

the formulas are more complicated and hopefully you might not need them at all :-) Refer to the bibliography if necessary.

calculating integrals: selecting the *best* value N (or h)

A) pick N such that h = (b-a)/N is "small enough"

- good for well-behaved functions; program is simpler too!
- but the error in "I" might still be big even for small values of h
- **or**, h might just be ***too*** small ***-don't overdo calculations!-***

B) start with a small value of N, re-evaluate I for 2N, 4N, 8N, etc, until the newest Integral differs from the previous one by less than a predetermined "Epsilon"

- slightly more complicated, but previous results of "**I**" can be used to calculate new results.

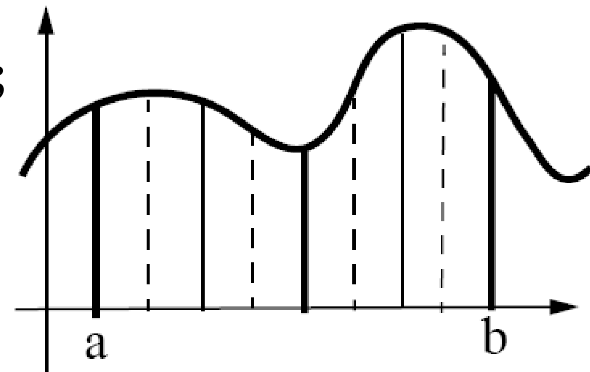
$\mathbf{I_1 = (b-a) * (f_a/2 + f_b/2\ );\ N = 1}$

$\mathbf{I_2 = I_1/2 + ((b-a)/2) * f((a+b)/2);\ N = 2}$

$\mathbf{I_4 = I_2/2 + ((b-a)/4) * (\ f((3a+b)/4) + f((a+3b)/4)\ );}$

$\mathbf{N = 4}$**, etc.**

continue until $\mathbf{abs(I_n - I_{2n}) < Epsilon}$



Ordinary Differential Equations (ODEs)

Any ODE can be rewritten in terms of a set of first order ODEs:
For example:

$$\frac{d^2 y}{dx^2} + q(x) \cdot \frac{dy}{dx} = r(x)$$

can be written as two separate equations:

$$\frac{dy}{dx} = z(x) \qquad \frac{dz}{dx} + q(x) \cdot z(x) = r(x)$$

The general problem therefore can be written in terms of N first order equations of the form:

$$\frac{dy_i}{dx} = f_i\left(x, y_1, y_2, \ldots, y_N\right), \quad i = 1, 2, \ldots, N$$

Of course, boundary conditions are needed in order to solve specific problems.

Ordinary Differential Equations (ODEs) (cont'd)

General Problem:

$$\frac{d}{dx} y(x) = f(x, y(x))$$

with known boundary condition $y_0 = y(x_0)$

- **first order approximation, or *Euler's method***

  a) write the equation in terms of finite differences:

$$\Delta y = (\Delta x) f(x, y)$$

  b) choose a step increment h so that $x_n = x_{n-1} + h$

  c) with $\mathbf{y_n} = \mathbf{y}(\mathbf{x}_n)$, use the following recursion relation:

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \mathbf{h} * \mathbf{f}(\mathbf{x}_n, \mathbf{y}_n) + \mathbf{O}(\mathbf{h}^2)$$

ODEs: higher order methods

- **2nd order Runge-Kutta method**

- use the derivative at the midpoint of the h-interval

$$\mathbf{k_1 = h * f(x_n, y_n)}$$
$$\mathbf{k_2 = h * f(x_n + h/2, y_n + k_1/2)}$$
$$\mathbf{y_{n+1} = y_n + k_2 + O(h^3)}$$

Note that the gain in accuracy is only one power of h, while **f(x,y)** has to be evaluated twice at each step. Not too good! :-(

- **4th order Runge-Kutta method**

- most often used, it requires 4 evaluations of the function f(x,y), but then the precision improves to $O(h^5)$ VERY GOOD! :-)

$$\mathbf{k_1 = h * f(x_n, y_n)}$$
$$\mathbf{k_2 = h * f(x_n + h/2, y_n + k_1/2)}$$
$$\mathbf{k_3 = h * f(x_n + h/2, y_n + k_2/2)}$$
$$\mathbf{k_4 = h * f(x_n + h, y_n + k_3)}$$

$$\mathbf{y_{n+1} = y_n + k_1/6 + k_2/3 + k_3/3 + k_4/6 + O(h^5)}$$

ODEs: choosing the *right* value of h

The value of **h should be modified** according to the rate of change of y, in order to improve accuracy and efficiency

- when the solution "dy" does not change too much, h should be increased to improve efficiency
- when "dy" varies rapidly, h must be reduced to ensure better numerical accuracy

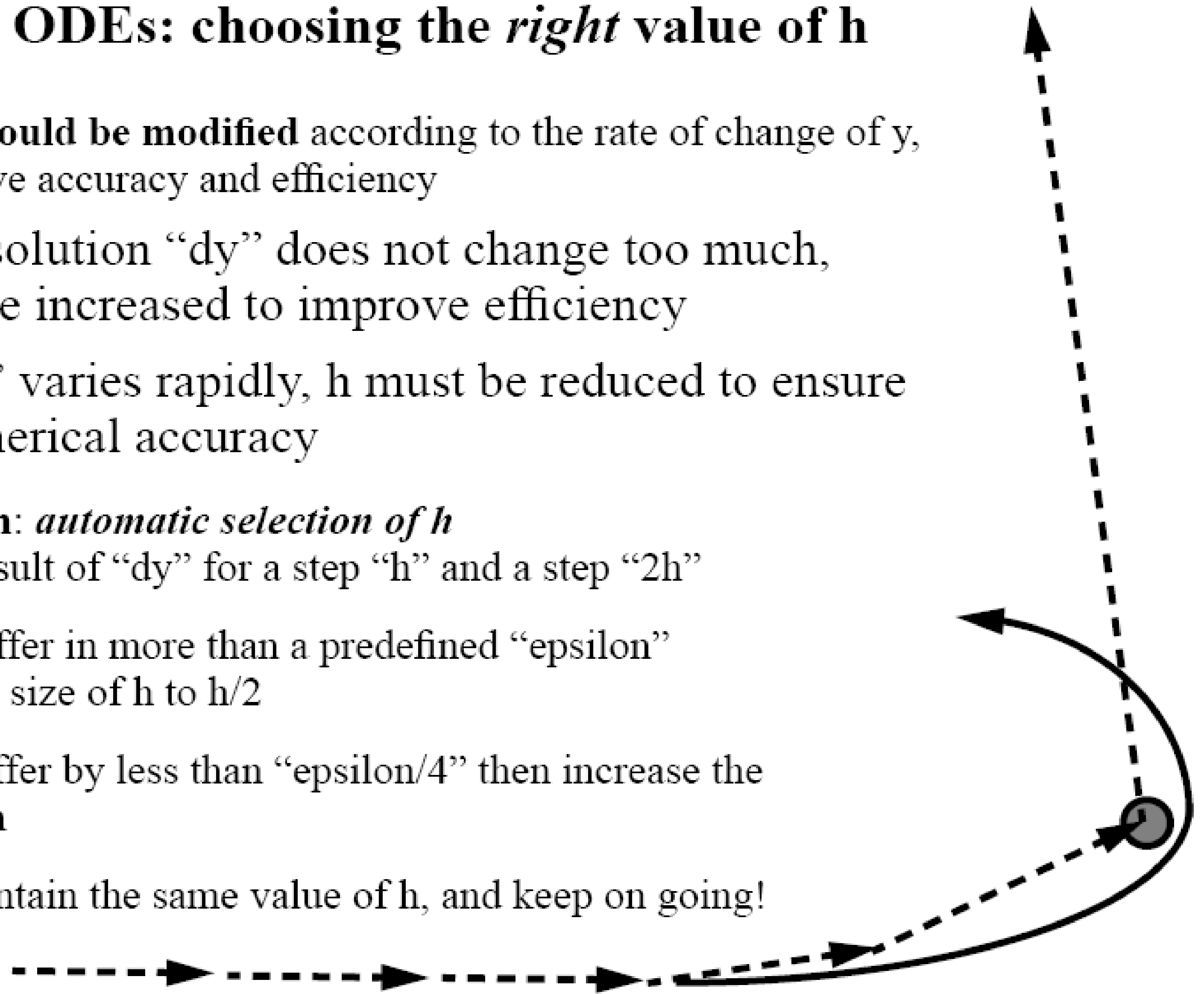
Simple algorithm: ***automatic selection of h***

a) compare the result of "dy" for a step "h" and a step "2h"

b) if the "dy"s differ in more than a predefined "epsilon" then reduce the size of h to h/2

c) if the "dy"s differ by less than "epsilon/4" then increase the value of h to 2h

d) otherwise, maintain the same value of h, and keep on going!



Issues of Parallel Computing

- Pros :
  - Solve problems faster
  - Solve larger problems
  - Solve more complex problems
- Cons :
  - Difficult to construct
  - Efficient parallel algorithm may need some thought
  - Cost of program development
- **Key issues:**
  1) **LOAD BALANCE - same amount of work for every processor**
  2) **LOCALITY - minimize communications among processors**
  3) **PORTABILITY – should work well on different computer systems**
  4) **SCALABILITY – performance scales well with increase in CPUs**
- **Performance depends on:**
  1) **Serial performance of the application**
  2) **Parallel efficiency of the application**
  3) **Mapping of data to parallel machines**
  4) **Effective use of data communication library**

Parallel Programming Approaches

- Data Parallel Languages: (data is distributed over the processors as arrays)
  - Entire arrays are manipulated A(1:100) = B(1:100) + C(1:100)
  - The compiler transforms this to instructions on the processors and data
  - e.g.: Fortran 90 (F90), High Performance Fortran (HPF), CoArray Fortran
- Message Passing Libraries
  - Programmer is responsible for data distribution, program synchronizations, and sending and receiving information (using message passing functions)
  - e.g.: Parallel Virtual Machine (PVM), Message Passing Interface (MPI)
- Automatically parallelizing compilers
  - Compilers analyze programs and parallelize (usually loops)
  - Easy to use, but with limited success
  - e.g. UPC (Unified Parallel C)

Improving Message Passing

- Safety first: Using blocking (or synchronous) message passing is much safer. You block everything until you are sure the message has arrived safely.
- However this is not always efficient. The program could be doing additional useful work while the message is being sent, or while it is waiting on another message.
- Care must be taken to ensure that the data to be sent, or the data to be received is not used until that data has actually been sent or received.

Processor-5	Processor-9
send(matrix-A to node 9,...) --->	recv(matrix-A from node 5,...)
--keep busy doing other things---	--keep busy doing other things---
--and more things -----	--and more things -----
probe(is matrix-A gone?)	--and more things -----
if not (block)	probe(has matrix-A arrived?)
-now you can modify matrix-A	if not (block)
---continue with code---	-now you can use matrix-A
	---continue with code---

Performance Evaluation of Codes

- Profiling of codes
  - Use profiling tools, (e.g. prof, fpmpi, pat_hwpc, -Mprof=func, -Ktrap=fp, ftn –rm, cc –hlist=m, PAT_RT_EXPERIMENT=`samp_cs_time`) to identify segments of code that take up most of the computing time

- Timing codes
  - Unfortunately many timers vary on different machines platforms
  - Shell command: > time *executable* ---> cpu time, elapsed time
  - Functions: etime, cpu_time, gettimeofday, getrusage, …

    should provide good timing resolution for sections of codes
- MPI timer
  - MPI_Wtime() ---> return seconds of elapse wall-clock time
  - MPI_Wtick() ----> return value of seconds between successive clock ticks

Communication Characteristics

Relatively slow communication vs. computation

- The bigger communication cost is in the "startup" or overhead
  - for example, a 40 usec (software) latency means that sending separate 1-byte messages --> 1byte/40us = 25 KBytes/sec !!
  - also hardware startup time is much smaller (~several nanosecs) than the software latency (~several microsecs)

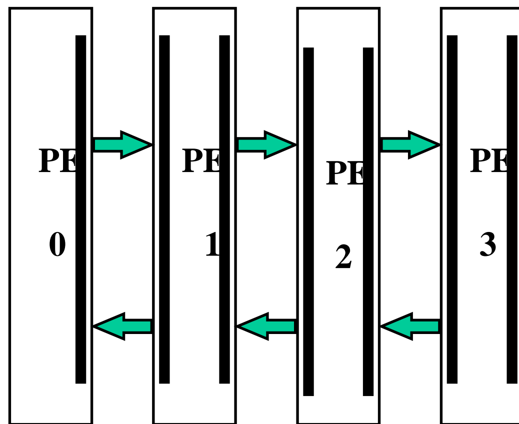
Thus always try to send few large messages rather than many small ones !

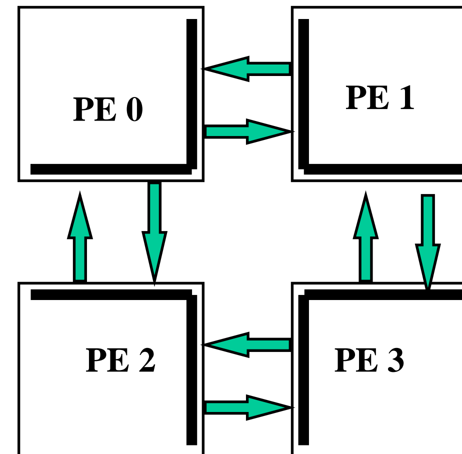
Bottom line: try to minimize the ratio of (# messages / # computations)

- Startup time and Bandwidth
  - Memory bandwidth, e.g
    - If 800MB/s peak => 100 MFLOPS peak if data resides in memory !!!
  - Parallel system bandwidth and latency
    - If 400 MB/s peak bandwidth => 50 MFLOPS peak if data reside in memory of other processor
    - 1 ms latency + a message of 20 MB => 49.5 MFLOPS peak
    - 1 ms latency + a message of 0.2 MB => 25 MFLOPS peak

I/O and Parallel I/O

- I/O can be a serious bottleneck for certain applications. The time to read/write data to disks could be an issue. But sometimes the shear size of the data file is a problem.
- Parallel I/O systems allow (in theory) the efficient manipulation of large files of 2 GigaBytes or greater from multiple nodes to multiple disks.
- Unfortunately, parallel I/O is only available on some architectures, and software is not always good. (MPI-2 has parallel MPI-IO on ROMIO implementation)
- Fortunately, on Cray XT3, 4, and 5's Kraken's Lustre filesystem has good parallel I/O with striping capability
- On the IBM SP systems with GPFS
- Using local disks /tmp for input output, if available; (NOT on Cray XTs!)





Improving Scientific Computing: the process

- 1.Write the program, or build it from previous codes, etc.
- 2. Debug your code (with optimization switches off)
- 3. Ensure mathematical correctness of the program!
- 4. Profile your code – determine where most of the computing time is spent
- 5. Optimize the algorithm, the data mapping, the communication, the I/O
- 6. Try out different combinations of compiler flags and/or compiler directives
- 7. Profile your code again
- 8. Re-examine blocks of code that consume the most execution time
- 9. Repeatedly apply various optimizations to such blocks
- 10. Rerun optimized code, compare performance, and start again until "satisfied".

Final thoughts: Strategies for Improved Performance

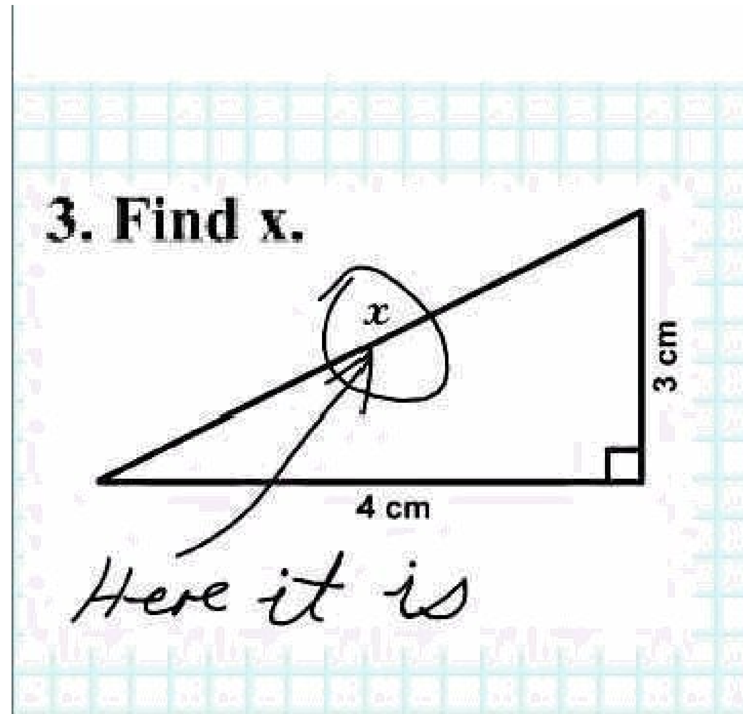
- Improving performance is a complex task, and the amount of time and effort put into it might not always be worth it.
- A certain trade-off must be reached between the developmental time and the "final" production run time.
- If you need to work on a previously existing code, then take the time to learn the details of its logic (if possible). Sometimes you might be better off rewriting the whole code directly in parallel!
- If you write the program from scratch, take some time to think about the different performance issues presented here and/or elsewhere.
- Examine benchmark results and know the limits of the computing platform

Finally: What else can be done?

- Practice, try new approaches, innovate, ask others
- Remember to concentrate only on subroutines worth improving
- Rethink the whole algorithm from scratch !?
- Remember to re-check the results for "correctness" (whenever possible!)
- Change parallel method (?), or change parallel machine (?)
- (ask someone else to do the calculations! ;-)

Food for thought:

Sometimes the obvious does not seem obvious:



(copied from an actual high-school math test)

Sources and Thanks

- **The material for this presentation comes from extensive personal experience, as well as from the following sources:**

  - many WWW sites
  - numerous interesting books
  - other workshops, conferences and meetings

Details of WWW addresses and book references can be obtained upon request.

Many thanks to:
- Kwai Wong, Research Scientist, JICS, for insightful discussions and for a few slides
- JICS graduate research assistants who tested some of the examples a while back

Questions? Comments?

Thank you!

The End

Addendum A
Some codes and sci-comp works developed by the presenter

- Phenogram distances for taxonomic studies of herpetological species..
- Filling and emptying of a 2-D porous media with liquids.
- Deconvolution of experimental data into sums of exponential functions, Prony.
- Heat distribution in non-homogeneous cylindric conductors.
- Time Fourier sequences of solar induced ionospheric activity.
- Dielectric relaxation of emulsions of oil/water at high frequencies.
- Electrostatic potentials of dielectric liquids under intense electric fields.
- Determining all real roots of a polynomial of degree n; symbolic math, Sturm.
- Board game of Othello-Reversi; (the computer won most times).
- Sphere packing graphical interfaces w/XView (for K.Stephenson)

- Surface Energy Distributions using Supercomputers.(w/Guiochon & B.Stanley)
- Dielectric relaxation as a series of Debye functions (R.Barchini)
- Parallel QR algorithm (ScaLAPACK) (J.Dongarra)
- Neural Network Simulations on Massively Parallel Computers; Applications in Chemical Physics (w/Noid & Sumpter)
- Parallel implementation of an Individually Based Algae Model (E.Miller)
- Parallelization of the Vehicle Routing Problem via Set Partioning (S.Wolf)

Addendum B
"Parallel" and Teaching Experience

Developed and ported codes, implemented and taught training workshops on parallel computer systems such as the:

- Thinking Machines CM5 (32 nodes)
- MasPar MP-2 (4096 CPUs)
- Intel iPSC/860 (128 nodes)
- Kendall Square KSR-1 (64 nodes)
- Intel Paragon (66 nodes)
- IBM SP-2 (16, 48, and 512 nodes)
- SGI Origin-2000 (32 nodes)
- IBM SP-3 (704 pwr3 nodes)

using PVM, MPI, HPF and specific systems languages.

Taught over 60 workshops, including Paragon workshops for Intel's Scalable Systems Division, at WPAFB, Ohio and at HKUST, Honk Kong; and 2 workshops selected for Supercomputing 99 (Maximizing your Megaflops) and for Supercomputing 2001 (Build your own parallel computing cluster now; for scientific computing).

Hired and supervised over 30 GRAs; member of over a dozen graduate committees (PhDs and/or MSs)

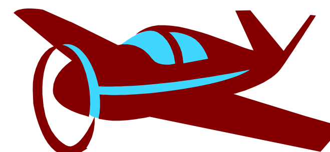
Taught >8 years of undergraduate level and graduate level Physics courses (Mechanics, Electromagnetism, Electrodynamics, Relativity, Computational Physics, etc).

Addendum C
Interesting (?) trivia about the presenter

- Born in one country, citizen of another, and spent most of his life in two other countries. Each of these four countries are in a different continent! (Burundi, Belgium, Argentina, USA; also lived in France (2 yrs) and Zaire (6 yrs)).
- Speaks, reads, and writes fluently English, French and Spanish.

- Discovered a new species of frog, which was then named after him as **Gastrotheca christiani.** It's a marsupial tree frog from northern Argentina, 1967.
- Was Vice President (2 years) and then President (2 years) of a mountain climbing club called Associacion Tucumana de Andinismo, 1970-74.

- Climbed several times over 20,000 ft in the Andes, as well as Mt. Hood (OR) and Mt. St. Helen (WA).
- Private Pilot Certificated, Single Engine Land, August 2005.



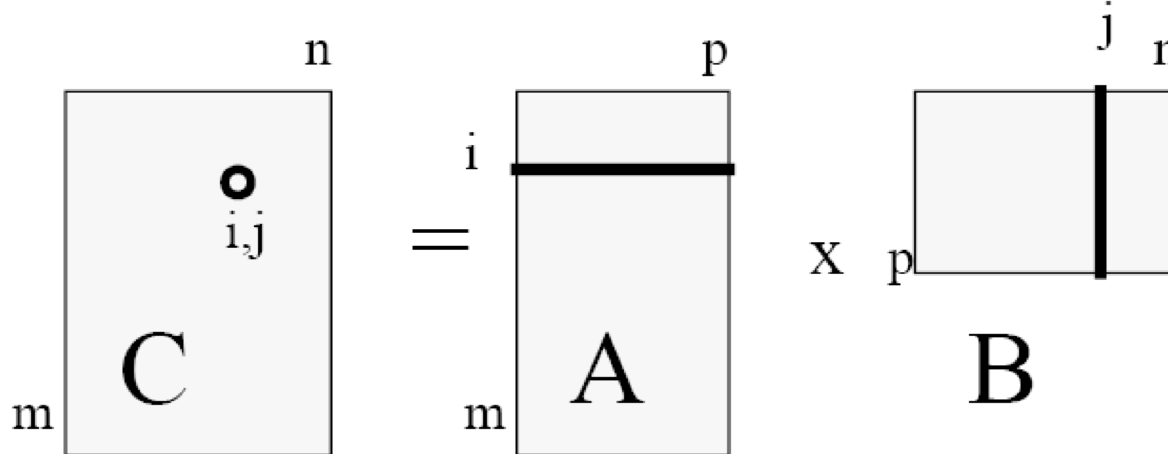
Math optimizations: EXAMPLE 1
Matrix Multiplication Problem

C = A x B or **C(m, n) = A(m, p) x B(p, n)**

or

$$c_{ij} = \sum_{k}^{p} a_{ik} \cdot b_{kj}$$

Number-of-flops = m . n . [p + (p - 1)] ==> N . N . (2 . N - 1) (if m=n=p=N)



example: REAL A(1000, 1000), B(1000, 1000), C(1000, 1000); **Nflops = 2.10^9**

Analysis of MM

(www.cs.berkeley.edu/ /~demmel/cs267_Spr99/)

- To quantify the analysis, a simple model of two levels of memory hierarchy, fast and slow, are used. All data initially resides in slow memory. Define

  m = number of memory references to slow memory needed just to read the input data from slow memory, and write the output data back

  f = number of floating point operations

  q = f/m = average number of flops per slow memory reference

- Hence, the higher the value of q , the more efficient the algorithm

  m = n^3 ---> read each column of B n times +

  n^2 ---> read each row of A for each I +

  2*n^2 ---> read/write each entry of C once --------> n^3 + 3* n^2

  f = 2* n^3

  q = f/m = (2* n^3) / (n^3 + 3* n^2) ~ 2

- Ideal value of q = n/2

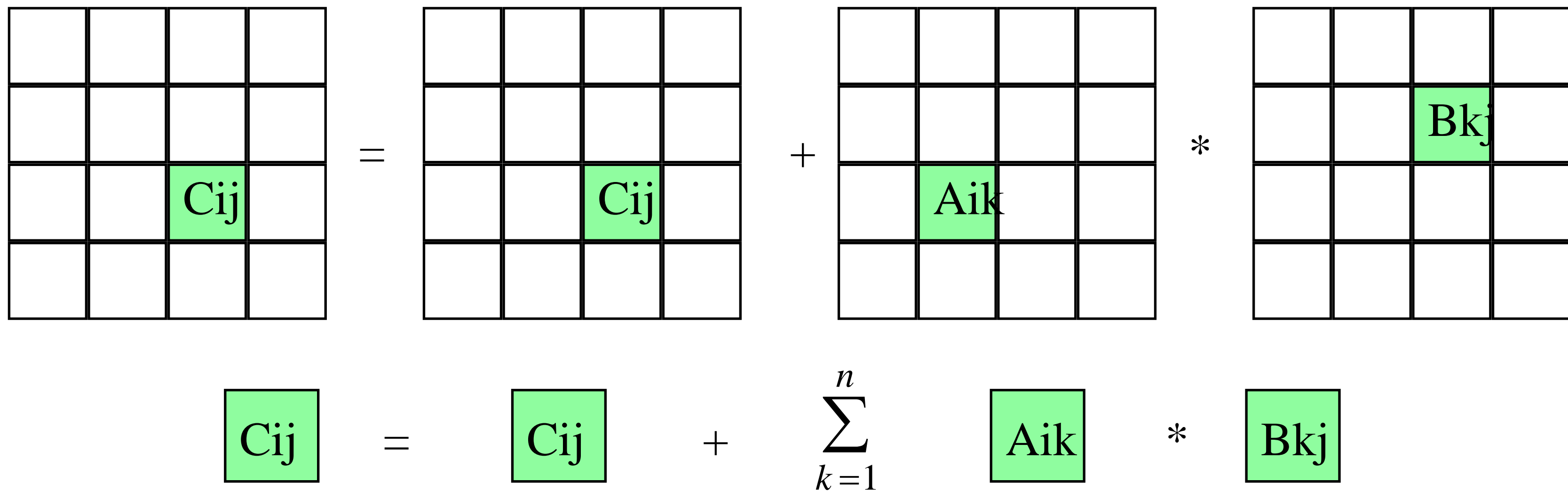
  ideal value of m = 4* n^2 ----> read each A(I,j), B(I,j), C(I,j) once,

  write each C(I,j) once

  hence, ideal value of q = f/m = n/2

Block MM

- **q = f/m = (2*n^3) / ((2*N + 2) * n^2) ~ n / N**
- **If N is equal to 1, the algorithm is ideal. Although, N is bounded by the amount of fast cache memory, N can however be chosen independently of the size n of the matrix.**
- **The optimal value of N = sqrt (size of fast memory / 3)**



Parallel MM

- Block Parallel MM : Map Matrix A and B to processors of 2D mesh

A(0,0)	A(0,1)	A(0,2)
A(1,0)	A(1,1)	A(1,2)
A(2,0)	A(2,1)	A(2,2)

*

B(0,0)	B(0,1)	B(0,2)
B(1,0)	B(1,1)	B(1,2)
B(2,0)	B(2,1)	B(2,2)

=

C(0,0)	C(0,1)	C(0,2)
C(1,0)	C(1,1)	C(1,2)
C(2,0)	C(2,1)	C(2,2)

- **Cannon Algorithm :**

1) Align Matrix A and Matrix B for computations. Shift A ij left by I steps. Shift Bij up by j steps. Perform the first multiplication.

2) Shift A ij left by 1 steps. Shift Bij up by 1 steps. Perform the second multiplication.

3) Shift A ij left by 1 steps. Shift Bij up by 1 steps. Perform the third multiplication.

A(0,0)	A(0,1)	A(0,2)
A(1,1)	A(1,2)	A(1,0)
A(2,2)	A(2,0)	A(2,1)

X

B(0,0)	B(1,1)	B(2,2)
B(1,0)	B(2,1)	B(0,2)
B(2,0)	B(0,1)	B(1,2)

=

C(0,0)		
	C(1,1)	
		C(2,2)

Nearest Neighbors: results for 5000 particles

run times in **seconds** on different computers
(using cc or gcc compilers)

	nn0	nn0,-O2	nn1	nn1,-O2	nn2	nn2,-O2	nn3	nn3,-O2
Sparc 2	369.3	82.7	177.2	47.9	92.95	29.8		
Sparc 5	161.2	49.3	77.8	18.8	42.3	11.8		
SP2-pwr2	15.2	13.4	13.5	11.5	6.49	3.01	4.34	2.76
Cheetah x10					14.6	12.2	12.5	12.6
Jaguar								
MasPar (4096procs)					2.42	1.54 (w/-Omax)		
MPI cluster 5 sparc5s					26.6	7.46		

Math performance – use libs

Review and modify this slide!?

Using Math Library Subroutines

BLAS, ESSL, LinPack, LAPack, ScaLAPACK, AZTEC, PETSc, PINAPL, etc.

- Generally the most important kernels for scientific computations
- Using these math libraries can often simplify coding.
- They are portable across different platforms
- They are usually finetuned to the specific hardware as well as to the sizes of the array variables that are sent to them; (block-parameters)
- Occasionally additional improvement can be obtained by "cleaning up" the math routines to fit the specific needs of your problem. This will reduce the overhead associated with all the special cases that these routines consider.