AMath 483/583 — Lecture 12

Outline:

- More about computer arithmetic
- Fortran optimization and compiler flags
- Parallel computing

Reading:

- Optimization flags: `http://gcc.gnu.org/onlinedocs/gcc-3.4.5/gcc/Optimize-Options.html`

  class notes: bibliography for general books on parallel programming

Excess precision

In Homework 3 some people noticed different small values reported when evaluating $f(x)$ for x very close to root.

Try compiling with gfortran flag `-ffloat-store`.

This forces variables to be written out of registers to cache before reusing.

Excess precision

In Homework 3 some people noticed different small values reported when evaluating $f(x)$ for x very close to root.

Try compiling with gfortran flag `-ffloat-store`.

This forces variables to be written out of registers to cache before reusing.

Sometimes registers have more precision than other memory to try to get a bit better accuracy.

Sometimes nice, but can destroy reproducibility.

Floating point real numbers

Base 10 scientific notation:

$$\texttt{0.2345e-18} = 0.2345 \times 10^{-18} = 0.000000000000000000002345$$

Mantissa: 0.2345, Exponent: -18

Floating point real numbers

Base 10 scientific notation:

$$\texttt{0.2345e-18} = 0.2345 \times 10^{-18} = 0.000000000000000002345$$

Mantissa: 0.2345, Exponent: -18

Binary floating point numbers:

Example: Mantissa: 0.101101, Exponent: -11011 means:

$$\begin{aligned} 0.101101 &= 1(2^{-1}) + 0(2^{-2}) + 1(2^{-3}) + 1(2^{-4}) + 0(2^{-5}) + 1(2^{-6}) \\ &= 0.703125 \ \text{(base 10)} \\ -11011 &= -1(2^4) + 1(2^3) + 0(2^2) + 1(2^1) + 1(2^0) = -27 \ \text{(base 10)} \end{aligned}$$

So the number is

$$0.703125 \times 2^{-27} \approx 5.2386894822120667 \times 10^{-9}$$

Floating point real numbers

Fortran:

real (kind=4): 4 bytes
    This used to be standard single precision real

real (kind=8): 8 bytes
    This used to be called double precision real

Python float datatype is 8 bytes.

8 bytes = 64 bits,

53 bits for mantissa and 11 bits for exponent (64 bits $=$ 8 bytes).

We can store 52 binary bits of precision.

$2^{-52} \approx 2.2 \times 10^{-16} \implies$ roughly 15 digits of precision.

Floating point real numbers (8 bytes)

Since $2^{-52} \approx 2.2 \times 10^{-16}$

this corresponds to roughly 15 digits of precision.

We can hope to get at most 15 correct digits in computations.

For example:

```
>>> from numpy import pi
>>> pi
3.1415926535897931

>>> 1000 * pi
3141.5926535897929
```

Note: storage and arithmetic is done in base 2
Converted to base 10 only when printed!

Absolute and relative error

Let $\hat{z} =$ exact answer to some problem,
$z^* =$ computed answer using some algorithm.

Absolute error: $|z^* - \hat{z}|$

Relative error: $\dfrac{|z^* - \hat{z}|}{|\hat{z}|}$

If $|\hat{z}| \approx 1$ these are roughly the same.

But in general relative error is a better measure of
how many correct digits in the answer:

$$\text{Relative error } \approx 10^{-k} \quad \Longrightarrow \quad \approx k \text{ correct digits.}$$

Precision of floating point

If x a real number then $fl(x)$ represents the closest floating point number.

Unless overflow or underflow occurs, this generally has relative error

$$\left| \frac{fl(x) - x}{x} \right| \leq \epsilon_m$$

where ϵ_m is Machine epsilon.

$\epsilon_m \approx 10^{-k} \implies$ about k correct digits.

8-byte double precision: $\epsilon_m \approx 2.22 \times 10^{-16}$.

Machine epsilon (for 8 byte reals)

```
>>> y = 1. + 3.e-16
>>> y
1.0000000000000002

>>> y - 1.
2.2204460492503131e-16
```

Machine epsilon is the distance between 1.0 and the next largest number that can be represented: $2^{-52} \approx 2.2204 \times 10^{-16}$

```
>>> y = 1 + 1e-16
>>> y
1.0

>>> y == 1
True
```

Catastrophic cancellation of nearly equal numbers

We generally don't need 16 digits in our solutions

But often need that many digits to get reliable results.

```
>>> from numpy import pi
>>> pi
3.1415926535897931

>>> y = pi * 1.e-10
>>> y
3.1415926535897934e-10

>>> z = 1. + y
>>> z
1.0000000003141594   # 15 digits correct in z

>>> z - 1.
3.1415936518897070e-10  # only 6 or 7 digits right!
```

Sample compiler optimizations

See: `http://gcc.gnu.org/onlinedocs/gcc-3.4.5/gcc/Optimize-Options.html`

for a list of many gcc optimization flags.

Often `-O2` or `-O3` flag is used to include many common optimizations.

Global common subexpresion elimination

-fgcse (or -O2, -O3) optimization flag will replace:

```
do i=1,n
    y(i) = 2.d0 * x(i) * pi
    enddo
```

by machine code equivalent of...

```
pi2 = 2.d0 * pi
do i=1,n
    y(i) = pi2 * x(i)
    enddo
```

Global common subexpresion elimination

-fgcse (or -O2, -O3) optimization flag will replace:

```
do i=1,n
    y(i) = 2.d0 * x(i) * pi
    enddo
```

by machine code equivalent of...

```
pi2 = 2.d0 * pi
do i=1,n
    y(i) = pi2 * x(i)
    enddo
```

Note: May give slightly different results because computer arithmetic is non-commutative!

Sample compiler optimization – inlining functions

-finline-functions (or -O3) optimization flag will replace function calls by the corresponding code inline:

E.g., in `$UWHPSC/codes/fortran/newton/newton.f90`, replace

```
! evaluate function and its derivative:
fx = f(x)
fxprime = fp(x)
```

by machine code equivalent of...

```
fx = x**2 - 4.d0
fxprime = 2.d0*x
```

Sample compiler optimization – inlining functions

-finline-functions (or -O3) optimization flag will replace function calls by the corresponding code inline:

E.g., in `$UWHPSC/codes/fortran/newton/newton.f90`, replace

```
! evaluate function and its derivative:
fx = f(x)
fxprime = fp(x)
```

by machine code equivalent of...

```
fx = x**2 - 4.d0
fxprime = 2.d0*x
```

Overhead of function call is avoided. Can make a big difference if $f(x)$ is evaluated in a loop over large array.

Manual code optimization

Often it is necessary to rethink the algorithm in order to optimize code.

"Premature optimization is the root of all evil" (Don Knuth)

Once code is working, determine which parts of code need to be improved and spend effort on these sections.

Use tools such as gprof to identify bottlenecks.

Block matrix multiply

Compute $C = AB$. Can partition into blocks:

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

where

$$C_{ij} = A_{i1}B_{1j} + A_{i2}B_{2j}$$

When blocks A_{11} and B_{11} are in cache can compute the $A_{11}B_{11}$ *part* of $C_{11} = A_{11}B_{11} + A_{12}B_{21}$

Block matrix multiply

Compute $C = AB$. Can partition into blocks:

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

where

$$C_{ij} = A_{i1}B_{1j} + A_{i2}B_{2j}$$

When blocks A_{11} and B_{11} are in cache can compute the $A_{11}B_{11}$ *part* of $C_{11} = A_{11}B_{11} + A_{12}B_{21}$

Might next bring in B_{12} and compute the $A_{11}B_{12}$ *part* of $C_{12} = A_{11}B_{12} + A_{12}B_{22}$

Matrix transpose

```
do j=1,n
    do i=1,n
        b(j,i) = a(i,j)
        enddo
    enddo
```

Accessing a by column but b by row.

Switching loop order $\Longrightarrow$ accessing a by row!

Better to do by blocks

$$\begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}^T = \begin{bmatrix} A_{11}^T & A_{21}^T \\ A_{12}^T & A_{22}^T \end{bmatrix}$$

Matrix transpose

Suppose stride s divides n. Then can rewrite as:

Strip mining:

```
do jj=1,n,s
    do j=jj,jj+s-1
        do ii=1,n,s
            do i=ii,ii+s-1
                b(j,i) = a(i,j)
```

Loop reordering:

```
do jj=1,n,s
    do ii=1,n,s
        do j=jj,jj+s-1
            do i=ii,ii+s-1
                b(j,i) = a(i,j)
```

Loops over blocks in outer loops, within block in inner loops.

CPU time vs. throughput

a, b each 1000×1000 matrices. Compare multiply, add

Compare time of `c = matmul(a,b)` vs. `c = a+b`.

Compare megaflops per second: `1e-6*nflops/(t2-t1)`.

```
Add: CPU time (sec):  0.00687200
 rate:    145.52 megaflop/sec

Multiply: CPU time (sec):  2.38393500  slower
 rate:    838.53 megaflop/sec          higher
```

CPU time vs. throughput

a, b each 1000×1000 matrices. Compare multiply, add

Compare time of `c = matmul(a,b)` vs. `c = a+b`.

Compare megaflops per second: `1e-6*nflops/(t2-t1)`.

```
Add: CPU time (sec):  0.00687200
 rate:    145.52 megaflop/sec

Multiply: CPU time (sec):  2.38393500   slower
 rate:    838.53 megaflop/sec           higher
```

For addition: `nflops = n**2` $= \mathcal{O}(n^2)$
For multiplication: `nflops = (2n-1)*n**2` $= \mathcal{O}(n^3)$,
More flops, but each element is used n times,
$\Longrightarrow$ More flops per memory access $\Longrightarrow$ higher rate.

Parallel Computing

- Basic concepts
- Shared vs. distributed memory
- OpenMP (shared)
- MPI (shared or distributed)

Some general references

See class notes: bibliography

Some general books...

P. S. Pacheco, *An Introduction to Parallel Programming*, Elsevier, 2011.

T. Rauber and G. Ruenger, *Parallel Programming For Multicore and Cluster Systems*, Springer, 2010.

C. Lin and L. Snyder, *Principles of Parallel Programming*, 2008.

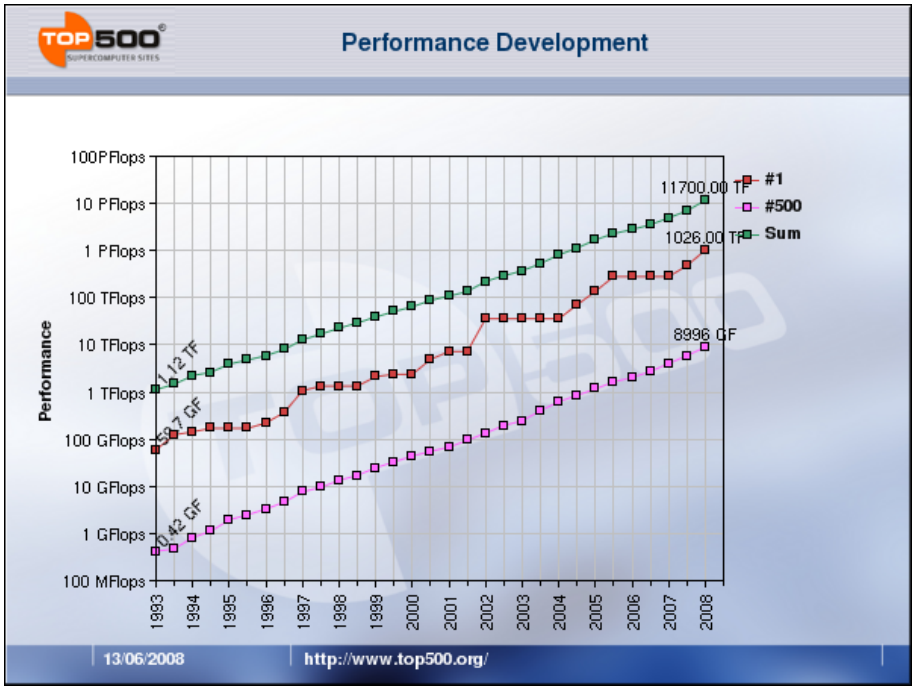
L. R. Scott, T. Clark, B. Bagheri, *Scientific Parallel Computing*, Princeton University Press, 2005.

Increasing speed

Moore's Law: Processor speed doubles every 18 months.

$\Longrightarrow$ factor of 1024 in 15 years.

Going forward: Number of cores doubles every 18 months.



Top: Total computing power of top 500 computers

Middle: #1 computer

Bottom: #500 computer

`http://www.top500.org`

Parallel processing

Shared memory:

All processors have access to the same memory.

Multicore chip: separate L1 caches, L2 might be shared.

Parallel processing

Shared memory:

All processors have access to the same memory.

Multicore chip: separate L1 caches, L2 might be shared.

Distributed memory:

Each processor has it's own memory and caches.

Transferring data between processors is slow.

E.g., clusters of computers, supercomputers

Parallel processing

Shared memory:

All processors have access to the same memory.

Multicore chip: separate L1 caches, L2 might be shared.

Distributed memory:

Each processor has it's own memory and caches.

Transferring data between processors is slow.

E.g., clusters of computers, supercomputers

General purpose GPU computing: (Graphical Processor Unit)

Parallel processing

Shared memory:

All processors have access to the same memory.

Multicore chip: separate L1 caches, L2 might be shared.

Distributed memory:

Each processor has it's own memory and caches.

Transferring data between processors is slow.

E.g., clusters of computers, supercomputers

General purpose GPU computing: (Graphical Processor Unit)

Hybrid: Often clusters of multicore/GPU machines!

Multi-thread computing

For example, multi-threaded program on dual-core computer.

Thread:

A thread of control: program code, program counter, call stack, small amount of thread-specific data (registers, L1 cache).

Shared memory and file system.

Threads may be spawned and destroyed as computation proceeds.

Languages like OpenMP.

POSIX Threads

Portable Operating System Intefrace

Standardized C language threads programming interface

For UNIX systems, this interface has been specified by the IEEE POSIX 1003.1c standard (1995).

Implementations adhering to this standard are referred to as POSIX threads, or Pthreads.

Multi-thread computing

Some issues:

Limited to modest number of cores when memory is shared.

Multiple threads have access to same data — convenient and fast.

Contention: But, need to make sure they don't conflict (e.g. two threads should not write to same location at same time).

Dependencies, synchronization: Need to make sure some operations are done in proper order!

May need cache coherence: If Thread 1 changes x in its private cache, other threads might need to see changed value.

Multi-process computing

A process is a thread that also has its own private address space.

Multiple processes are often running on a single computer (e.g. different independent programs).

For distributed memory parallel computers, a single computation must be tackled with multiple processes because of memory layout.

Larger cost in creating and destroying processes.

Greater latency in sharing data.

Multi-process computing

A process is a thread that also has its own private address space.

Multiple processes are often running on a single computer (e.g. different independent programs).

For distributed memory parallel computers, a single computation must be tackled with multiple processes because of memory layout.

Larger cost in creating and destroying processes.

Greater latency in sharing data.

Processes communicate by passing messages.

Languages like MPI — Message Passing Interface.

Multi-process computing with distributed memory

Some issues:

Often more complicated to program.

High cost of data communication between processes.
Want to maximize processing on local data relative to communication with other processes.

Often need to partition problem domain into subdomains,
(e.g. domain decomposition for PDEs)

Generally requires coarse grain parallelism.