Application Tuning

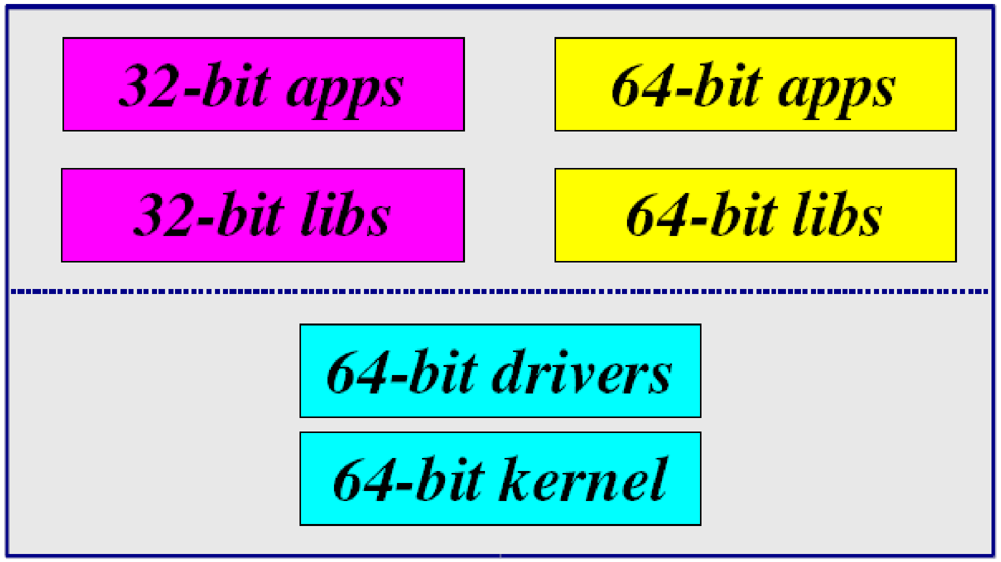
Selected Topics

Application Tuning

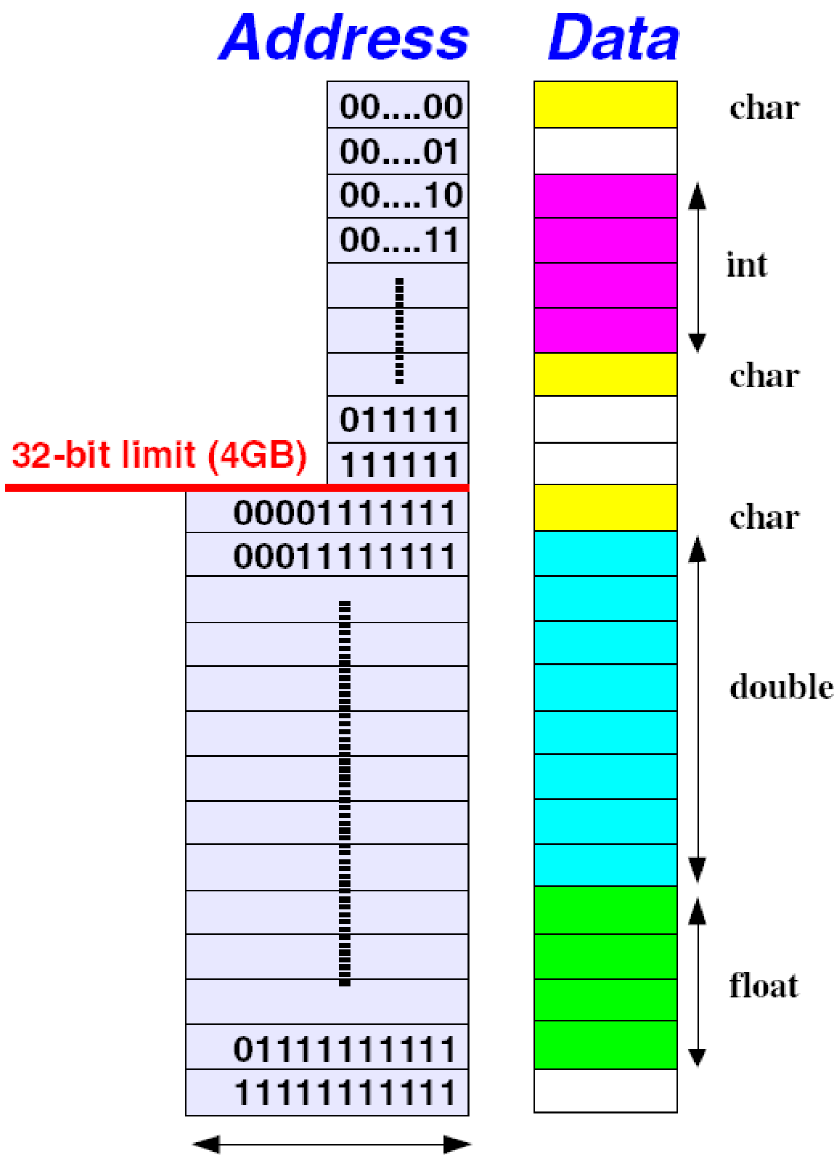
- Selected Topics:
  - 32- vs 64-bit
  - binary data portability
  - floating point numbers and IEEE 754
    - compiler options
    - case studies
  - large pages
- Summary

32-bit vs 64-bit issues

- 64-bit operating systems
- Implication: The address space of a single application can be larger than 4 GB



32-bit vs 64-bit issues



- Addresses ≠ Data
- An 'n'-byte data type fills always n bytes in memory (byte addressable)
- I.e. the next element is n bytes further in memory
- This increment is not related to the size of the addresses (32-bit or 64-bit)

32-bit vs 64-bit issues

C data type	ILP32 (bits)	LP64 (bits)
char	8	same
short	16	same
int	32	same
long	32	64
long long	64	same
pointer	32	64
enum	32	same
float	32	same
double	64	same
long double	128	same

UNIX and Linux support LP64; Windows 64-bit uses LLP64, where long stays 32 bits

(p)ldd and LD_LIBRARY_PATH

How to check which shared-libraries are loaded?

- Static check: use the ldd command
  - `$ ldd executable`
- Dynamic check: use pldd on the PID
  - `$ pldd pid`
  - Solaris only
  - there are scripts available for Linux as well

(p)ldd and LD_LIBRARY_PATH

How to change the search path for dynamic libraries?

- Use LD_LIBRARY_PATH – but use it with care!
- Solaris can distinguish between 32- and 64-bit:
  - LD_LIBRARY_PATH – common
  - LD_LIBRARY_PATH_32 – for 32-bit apps
  - LD_LIBRARY_PATH_64 – for 64-bit apps
- Linux: only one setting !!!

(p)ldd and LD_LIBRARY_PATH

Best practice:

- Compile the path into your application:
  - Sun Studio: -R <path_to_lib>
  - GCC: -Wl,-rpath <path_to_lib>
  - ld.so.1 will then use this path
- Avoid LD_LIBRARY_PATH in your shell environment – use a wrapper script for the application

Binary data storage

- Storing your data in binary format
- Advantages:
  - compact
  - fast
  - no loss of precision
- Drawbacks:
  - not "human readable"
  - data analysis more complicated
  - and ...

Binary data storage

- Example: integer 0x12345678 (hexadecimal)

```
value = 0x12345678;          // 305419896
printf("%d\n", value);
fwrite(&value, sizeof(value), 1, fptr);
```

- Write it ...
  - ... on i386:

```
305419896
Architecture: i386
Value written to endian_i386.dat.
```

  - ... on SPARC:

```
305419896
Architecture: sparc
Value written to endian_sparc.dat.
```

Binary data storage

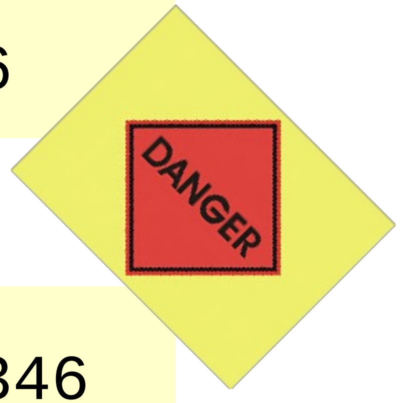
- Read it:

```
fread(&value, sizeof(value), 1, fptr);
printf("%d\n", value);
```

- on i386 data from i386:

```
Architecture: i386
Read from endian_i386.dat: 305419896
```

- on i386 data from SPARC:



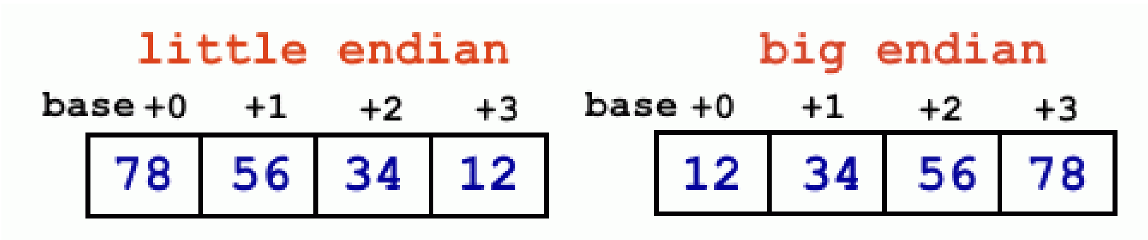

```
Architecture: i386
Read from endian_sparc.dat: 2018915346
```

Little Endian vs Big Endian

- The order in which the bits are interpreted has not been standardized!
- Two 'popular' formats in use
  - Big Endian – SPARC, PowerPC, ...
  - Little Endian – Intel x86, AMD64, ...
- This is an issue when using the same binary data file on both platforms ...

Little Endian vs Big Endian

- Example: integer 0x12345678 (hexadecimal)



- Check with 'od' command:

```
$ od -x endian_sparc.dat
0000000 1234 5678
0000004

$ od -x endian_i386.dat
0000000 7856 3412
0000004
```

Little Endian vs Big Endian

- This is something you should be aware of when working with binary data!
- Tools:
  - Sun Fortran: -xfilebyteorder option
  - Portland Fortran compiler
  - swab() subroutine (low level)
  - ...

Floating point numbers & IEEE 754

- Remember: -fast expands to a set of options, and two of them are:
  - -fns=yes: faster – but non-standard – handling of floating-point arithmetic exeptions and gradual underflow (small numbers)
  - -fsimple=2: aggressive floating-point optimizations
- If your code requires to follow strictly the IEEE Standard for Binary Floating Point Arithmetic (IEEE 754), you can use:
  - -fast -fns=no -fsimple=0 (or -fsimple=1)

Floating point numbers & IEEE 754

Effects of -fsimple:

- compiled with -fast -xrestrict -fsimple=0:

```
void
divvec(int n, double div, double *a, double *b) {

    int i;

Source loop below has tag L1
L-1 scheduled with steady-state cycle count = 17
L-1 has 1 loads, 1 stores, 2 prefetches, 0 FPadds,
        0 FPmuls, and 1 FPdivs per iteration
    for(i = 0; i < n; i++)
        b[i] = a[i]/div;
}
```

Floating point numbers & IEEE 754

Effects of -fsimple:

- compiled with -fast -xrestrict -fsimple=2:

```
 1. void
 2. divvec(int n, double div, double *a, double *b) {
 3.
 4.     int i;
 5.
Source loop below has tag L1
L-1 scheduled with steady-state cycle count = 1
L-1 unrolled 8 times
L-1 has 1 loads, 1 stores, 4 prefetches, 0 FPadds,
        1 FPmuls, and 0 FPdivs per iteration
 6.     for(i = 0; i < n; i++)
 7.         b[i] = a[i]/div;
 8. }
```

Floating point numbers & IEEE 754

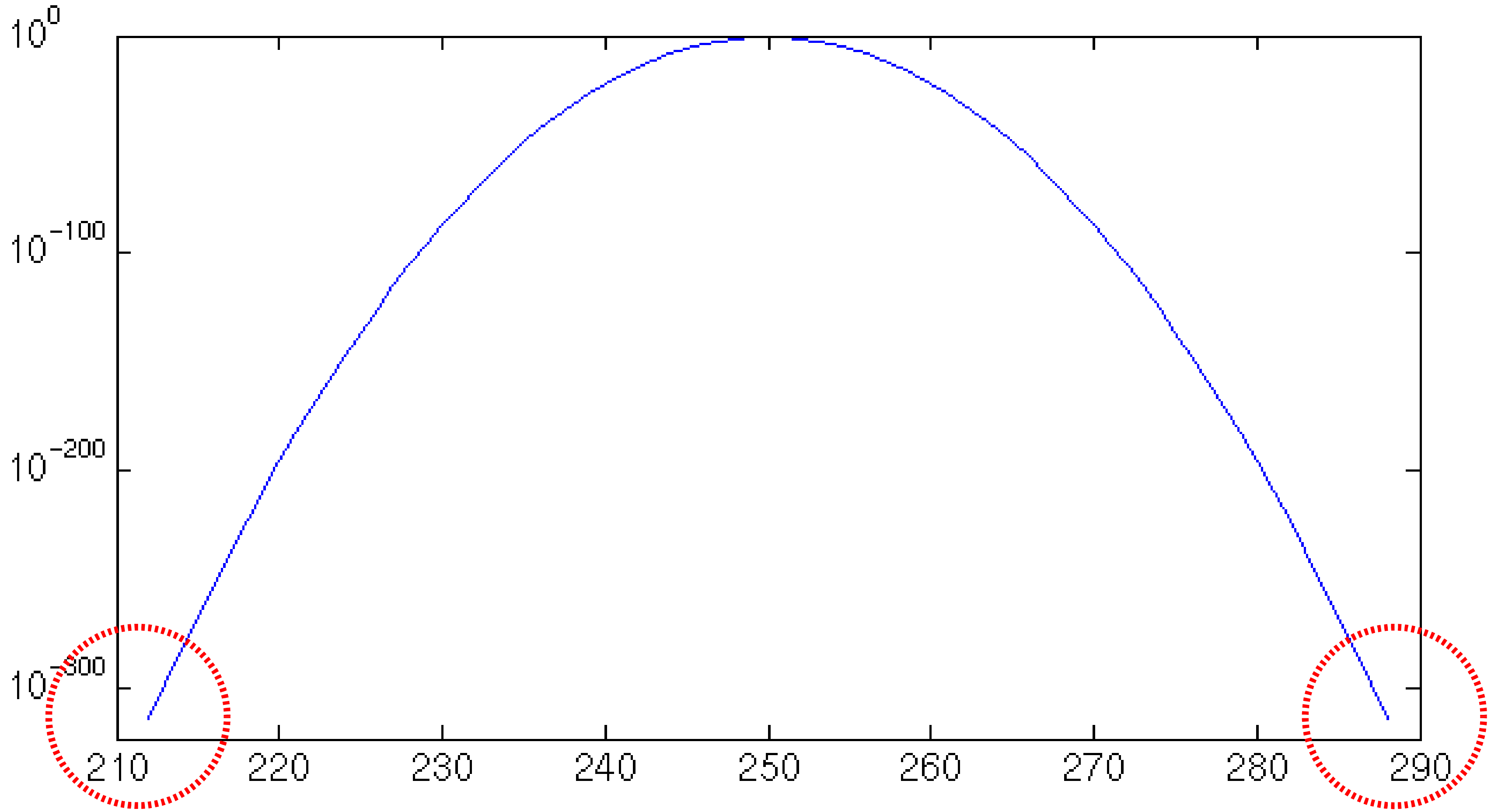
Effects of -fns=[yes|no] - a case study:

- Multiplying a Toeplitz matrix with a random matrix in Matlab took about 20 times longer than the product of two equally sized random matrices (on SPARC).
- This didn't happen on the Linux/Intel platform – here both operations took approximately the same time.
- An investigation of the structure of the Toeplitz matrix showed, that it had a large number of entries with *subnormal* numbers, i.e. numbers smaller than 10E-300.

Application Tuning

Floating point numbers & IEEE 754

One slice of the Toeplitz matrix:



Floating point numbers & IEEE 754

- What is going wrong here?
- Explanation:
  - The operations with the subnormal numbers result in lots of gradual underflows, every one causing a hardware trap on the SPARC platform. Those traps are really expensive (pipeline flushes, etc).
- Why's that?
  - Matlab on SPARC is compiled without the optimization option -fns that flushes those small numbers to zero.

Floating point numbers & IEEE 754

- Runtime of the Matlab native version with a 500x500 Toeplitz matrix: 14.45 secs
- Used the Matlab compiler mcc (mcc calls cc from Sun Studio) with the right optimization option (-fns=yes) in the mbuildopt.sh file.
- The runtime of the same example was reduced to 0.72 secs – a speed-up of 20x.
- The results of both versions are numerically identical!

Floating point numbers & IEEE 754

Another gradual underflow example:

- Cholesky factorization of a sparse matrix
- runtime: 90+ secs (39 secs user, 50 secs system)
- this example suffered from gradual underflows
- no possibility to recompile
- solution: add a small number (1e-12) to all matrix elements
- new runtime: < 9 secs – no system time overhead!

Floating point numbers & IEEE 754

- Events that can cause (hardware) traps:
  - division by zero
  - working with NaNs (Not A Number) – but some applications rely on that, e.g. for missing data points
  - gradual underflow
- Those traps can be a performance killer!
- BTW: Adobe Flash's floating data type initializes the value to NaN!

Large pages

- The TLB (or address cache) revisited:
  - mapping capacity: # TLB entries * page size
  - example (US-IV): 512 entries of 8 kB => 4MB
- Use large pages (if supported) if your application suffers from many TLB misses:
  - recompile & link with -xpagesize (Sun Studio)
  - Solaris:
    - ppgsz command (man ppgsz)
    - MPSS preload library (man mpss.so.1)

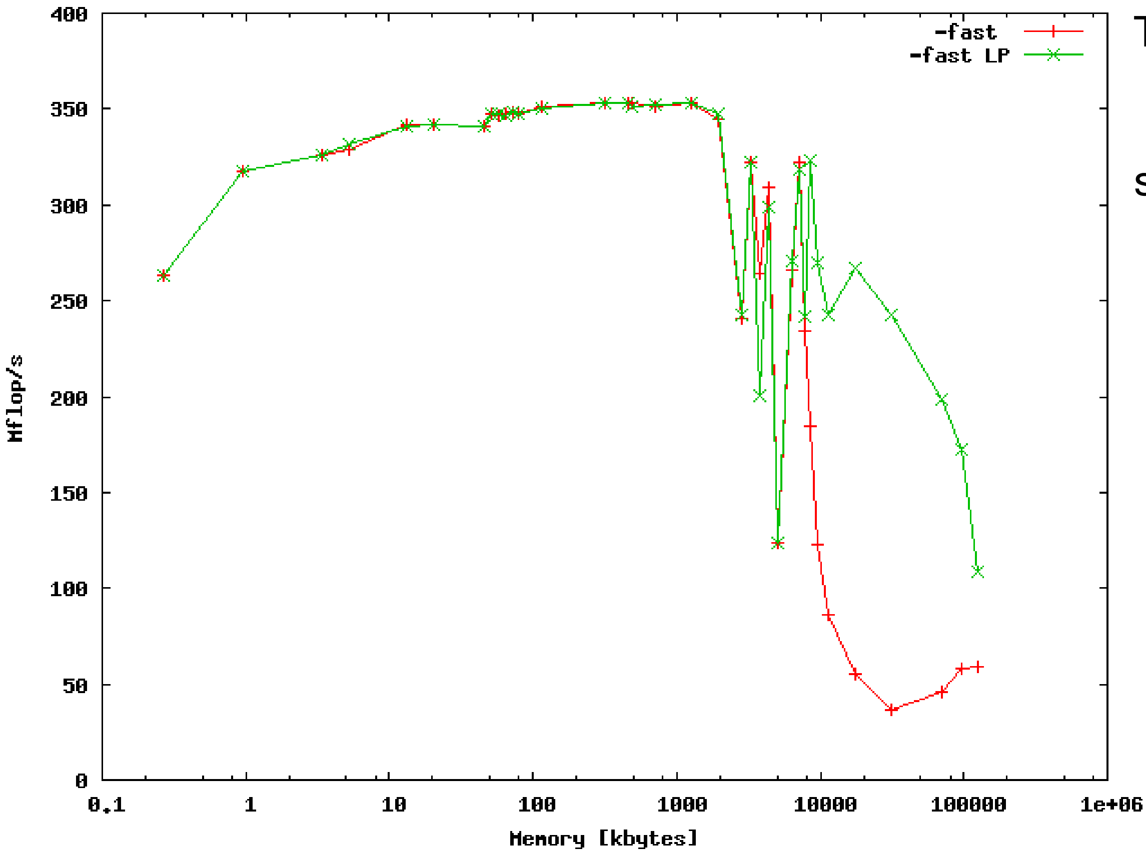
Large pages

- Changing the pagesize:
  - use one of the mechanism above
  - e.g. set pagesize to 4MB
  - mapping capacity: 512 entries of 4MB => 2GB!
- Check: with the 'pmap -sx <PID>' command
  - look for 'heap', 'anon' and 'stack' in the output
  - example: pmap -sx 12345 | egrep 'heap|stack|anon'
  - Note: although large pages are supported, they might not be available at run time!

Large pages

Effect of large pages: Matrix times vector

- column version (bad version)



Two runs:
  a) matvec
  b) ppgsz -o stack=4M,heap=4M matvec
same binary – no recompilation!

US-VI+ @ 1800 MHz
L1 : 64 kB
L2 : 2 MB
L3 : 32 MB
Peak : 3.6 Gflop/s

Large pages in Linux

- In Linux 'large pages' are called 'huge pages'
- This seems still under development
- Check the availability: 'cat /proc/meminfo'
- look for hugetlbpage.txt in the kernel sources

Summary

- You have now heard about
  - tuning techniques
  - tools: compilers, analysis tools
  - libraries
  - other performance parameters
  - debuggers: try Totalview
- Now you have to apply that and get experience!
- But never forget:

Correct code has the highest priority – not speed!