

Timing Considerations

- Needs to time operation as used
 - Most operations used differently by varying applications
 - Therefore, need to be able vary size/cache/align dynamically
 - Most accurate timings are platform-specific
 - Cycle-accurate clocks are arch-specific
 - Without above, must repeat calls with flushing
 - * Can get better, but not perfect in arch-indep manner
 - Timers need to work on all platforms, and be usable by non-root users on normally loaded machines
- ⇒ As much as possible, have platform options, but make work everywhere, and have option of cpu-time

Typical Performance Curves

No Flushing

- Dropoffs at cache edges
- Small problems faster than large
- Strong variance at beginning
- lesser variance at end

With Flushing

- Smooth curve towards asymptotic peak
 - Larger problems always faster than small
- Strong variance at beginning
- lesser variance at end

Using cycle-accurate timers

- CPUs run at Ghz speeds
 - Almost all cycle counts need 64 bit numbers
 - Most default sizeof(int) = 32 bit
 - Cycle counts therefore long long on most machines
 - Float doubles cannot hold 64 bit integer with precision, must either:
 1. Have timer shave most significant bits
 2. Do timing and transform to seconds in individual steps
- ⇒ Have similar problem with naive gettimeofday usage

Getting Standard Walitime

Boneheaded approach:

```
double my_time()
{
    struct timeval tp;
    gettimeofday(&tp, NULL);
    return(tp.tv_sec +
           tp.tv_usec/1000000.0);
}
```

- `tv_sec`: seconds since Jan 1, 1970
- $35\text{years} \times 365\text{days} \times 24\text{hours} \times 3600\text{sec} = 1.1e9$

⇒ Probably just rounded off all your microsec

Narrowed range:

```
double my_time() {
    struct timeval tp;
    static long start=0, startu;
    gettimeofday(&tp, NULL);
    if (start) {
        return(tp.tv_sec-start +
               (tp.tv_usec-startu)*1.0e-6)
    }
    else {
        start = tp.tv_sec;
        startu = tp.tv_usec;
        return(0.0);
    }
}
```

64-bit Approach for Standard Walltime

```
long long my_time(void)
{
    struct timeval tp;
    long long lret;
    gettimeofday(&tp, NULL);
    lret = tp.tv_sec;
    lret *= 1000000;
    lret += tp.tv_usec;
    return(lret);
}

double ClickToSec(long long clicks)
{
    return(clicks*1.0e-6);
}
```

- Makes timer lighter weight (assuming native 64 bit)
- Type conversion happens only once
- Any loss of precision happens only between start and stop, not beginning of timing run

```
long long t0, t1;

t0 = my_time();
sum = vecsum(N, X, Y);
t1 = my_time();
return(ClickToSec(t1-t0));
```

Timing with Cache Flushing

- **Basic idea:** After init of data, read/write $CS * \text{assoc}$ area of mem to flush cache.

Doesn't work:

- Flush before all calls, subtract out flush cost

Works for one invocation:

- Read/write big data before first call, after kernel init
- ISA-level cache flush

Work for multiple invocation(kind):

- Move operands around in $N + CS * \text{assoc}$ area

Cache Flushing with Single Invocation Timings

- `cs`: Cache size in elements
- `flush`: flushing workspace
- Read of contig flush forces `X` and `Y` out of cache

```
cs = cacheKB * (1024/sizeof(double))
flush = calloc(cs, sizeof(double))
... init operands ...
for (i=0; i<cs; i++)
    tmp += flush[i];
t0 = my_time();
sum = vecsum(N, X, Y);
t1 = my_time() - t0;
```

Cache-flushed Multiple Invocation DAXPY Timer

- DAXPY: $y \leftarrow \alpha x + y$
- cs : Cache size in elements
- nb : size of 1 invoc data block
- $nblk$: # of blks to flush cs
- Nt : Total elts in all blks
- Init self-flushes – might add additional read tho
- Go backwards to fool prefetch units
- Change sign of α to avoid overflow

Misc Timing Tricks

- Touch all mem before starting timings
 - OS zeros all newly allocated pages from sys
 - Many OSes use lazy zeroing
- Preload kernel into i-cache wt dummy arrays:
 - Small affect, may be ignorable with multiple samples

```
double xx[12], yy[12];
.... cache flush ...
daxpy(12, 1.0, xx, yy);
t0 = my_time();
daxpy(N, alpha, x, y);
t1 = my_time() - t0;
```

Sanity Checks for Timings

- $0 < mf < peak$?
- Different answers on diff archs?
- Speed goes up on faster machines?
- Speed changes with differing compile flags?
- Time drops by 1 sec when you add `sleep(1)`?
- Timer/kernel fail to compile when you add syntax error?
- Do curves match typical with & without cache flushing?
- Without flushing, do you see cacheedges where expected?
- For optimality:
 - Can you beat best compiler/flag combo?
 - Can you beat best known numbers?
 - Is number in max range for reuse?