# IN3200/IN4200: Chapter 2
# Basic optimization techniques for serial code

Textbook: Hager & Wellein, *Introduction to High Performance Computing for Scientists and Engineers*

# Objectives of Chapter 2

- "Common sense" and simple optimization strategies for serial code
- (Data access optimization will be discussed in Chapter 3)
- The role of compilers
- Basics of performance profiling

# “Common sense” optimizations

Very simple code changes can sometimes lead to significant performance boost.

The most important “common sense” principle: **avoiding performance pitfalls**!

# Do less work; example 1

Example: assume `A` is an array of numerical values, and a prescribed threshold value: `threshold_value`.

```
int flag = 0;
for (i=0; i<N; i++) {
  if ( some_function(A[i]) < threshold_value )
    flag = 1;
}
```

Improvement: leave the loop as soon as `flag` becomes `1`.

```
int flag = 0;
for (i=0; i<N; i++) {
  if ( some_function(A[i]) < threshold_value ) {
    flag = 1;
    break;
  }
}
```

# Do less work; example 2

```
for (i=0; i<500; i++)
  for (j=0; j<80; j++)
    for (k=0; k<4; k++)
      a[i][j][k] = a[i][j][k] + b[i][j][k]*c[i][j][k];
```

How many times is the `k`-indexed loop executed? And how many times for the `j`-indexed loop?

# Do less work; example 2 (cont'd)

If the 3D arrays `a`, `b` and `c` have **contiguous** memory storage for all their values, then we can re-code as follows:

```
double *a_ptr = a[0][0];
double *b_ptr = b[0][0];
double *c_ptr = c[0][0];

for (i=0; i<(500*80*4); i++)
  a_ptr[i] = a_ptr[i] + b_ptr[i]*c_ptr[i];
```

This technique is called *loop collapsing*. The main motivation is to reduce loop overhead, may also help other (compiler-supported) optimizations.

# Do less work; example 3

```
for (i=0; i<ARRAY_SIZE; i++) {
  a[i] = 0.;
  for (j=0; j<ARRAY_SIZE; j++)
    a[i] = a[i] + b[j]*d[j]*c[i];
}
```

Observation: `c[i]` is independent of the `j`-indexed loop.

# Do less work; example 3 (cont'd)

Improvement:

```
for (i=0; i<ARRAY_SIZE; i++) {
  a[i] = 0.;
  for (j=0; j<ARRAY_SIZE; j++)
    a[i] = a[i] + b[j]*d[j];
  a[i] = a[i]*c[i];
}
```

Can we improve further?

# Do less work; example 3 (further simplification)

There is a common factor:
`b[0]*d[0]+b[1]*d[1]+...+b[ARRAY_SIZE-1]*d[ARRAY_SIZE-1]`
which is unnecessarily re-computed in every `i` iteration!

```
t = 0.;
for (j=0; j<ARRAY_SIZE; j++)
  t = t + b[j]*d[j];

for (i=0; i<ARRAY_SIZE; i++)
  a[i] = t*c[i];
```

This technique is called *loop factoring* or *elimination of common subexpressions*.

# Another example of common subexpression elimination

```
for (i=0; i<N; i++)
  A[i] = A[i] + s + r*sin(x);
```

$$\Downarrow$$

```
tmp = s + r*sin(x);
for (i=0; i<N; i++)
  A[i] = A[i] + tmp;
```

# Avoid expensive operations!

Special math functions (such as trigonometric, exponential and logarithmic functions) are usually very costly to compute.

An example from simulating non-equilibrium spins:

```
for (i=1; i<Nx-1; i++)
  for (j=1; j<Ny-1; j++)
     for (k=1; k<Nz-1; k++) {
      iL = spin_orientation[i-1][j][k];
      iR = spin_orientation[i+1][j][k];
      iS = spin_orientation[i][j-1][k];
      iN = spin_orientation[i][j+1][k];
      iO = spin_orientation[i][j][k-1];
      iU = spin_orientation[i][j][k+1];
      edelz = iL+iR+iS+iN+iO+iU;
      body_force[i][j][k] = 0.5*(1.0+tanh(edelz/tt));
    }
```

# Example continued

If the values of `iL, iR, iS, iN, iO, iU` can only be $-1$ or $+1$, then the value of `edelz` (which is the sum of `iL, iR, iS, iN, iO, iU`) can only be $-6, -4, -2, 0, 2, 4, 6$.

If `tt` is a constant, then we can create a lookup table:

```
double tanh_table[13];
for (i=0; i<=12; i+=2)
  tanh_table[i] = 0.5*(1.0+tanh((i-6)/tt));
```

$$\Downarrow$$

```
for (i=1; i<Nx-1; i++)
  for (j=1; j<Ny-1; j++)
     for (k=1; k<Nz-1; k++) {
       ....
       edelz = iL+iR+iS+iN+iO+iU;
       body_force[i][j][k] = tanh_table[edelz+6];
    }
```

# Strength reduction

```
for (i=0; i<N; i++)
  y[i] = pow(x[i],3)/s;
```

$$\Downarrow$$

```
double inverse_s = 1.0/s;
for (i=0; i<N; i++)
  y[i] = x[i]*x[i]*x[i]*inverse_s;
```

# Strength reduction (another example)

```
for (i=0; i<N; i++)
  y[i] = a*pow(x[i],4)+b*pow(x[i],3)+c*pow(x[i],2)
        +d*pow(x[i],1)+e;
```

$$\Downarrow$$

```
for (i=0; i<N; i++)
  y[i] = (((a*x[i]+b)*x[i]+c)*x[i]+d)*x[i]+e;
```

Use of Horner's rule of polynomial evaluation:

$$ax^4 + bx^3 + cx^2 + dx + e = (((ax + b)x + c)x + d)x + e$$

# Shrinking the work set!

The *work set* of a code is the amount of memory it uses (or touches), also called *memory footprint*.

In general, shrinking the work set (if possible) is a good thing for performance, because it raises the probability of cache hit.

One example: The `spin_orientation` array should store values of type `char` instead of type `int`. (A factor of 4 in the difference of memory footprint.)

# Avoiding branches

"Tight" loops: few operations per iteration, typically optimized by compiler using some form of pipelining. In case of conditional branches in the loop body, the compiler optimization will easily fail.

```
for (j=0; j<N; j++)
  for (i=0; i<N; i++) {
    if (i>j)
      sign = 1.0;
    else if (i<j)
      sign = -1.0;
    else
      sign = 0.0;

    C[j] = C[j] + sign * A[j][i] * B[i];
  }
```

# Avoiding branches (cont'd)

```
for (j=0; j<N-1; j++)
  for (i=j+1; i<N; i++)
    C[j] = C[j] + A[j][i] * B[i];

for (j=1; j<N; j++)
  for (i=0; i<j; i++)
    C[j] = C[j] - A[j][i] * B[i];
  }
```

We have got rid of the if-tests completely!

# Another example of avoiding branches

```
for (i=0; i<n; i++) {
  if (i==0)
    a[i] = b[i+1]-b[i];
  else if (i==n-1)
    a[i] = b[i]-b[i-1];
  else
    a[i] = b[i+1]-b[i-1];
}
```

# Another example of avoid branches (cont'd)

Using the technique of *loop peeling*, we can re-code as follows:

```
a[0] = b[1]-b[0];
for (i=1; i<n-1; i++)
  a[i] = b[i+1]-b[i-1];
a[n-1] = b[n-1]-b[n-2];
```

# Yet anothe example of avoiding branches

```
for (i=0; i<n; i++) {
  if (j>0)
    x[i] = x[i] + 1;
  else
    x[i] = 0;
}
```

$$\Downarrow$$

```
if (j>0)
  for (i=0; i<n; i++)
    x[i] = x[i] + 1;
else
  for (i=0; i<n; i++)
    x[i] = 0;
```

# Using SIMD instructions

A "vectorizable" loop can potentially run faster if multiple operations can be performed with a single instruction.

Using SIMD instructions, register-to-register operations will be greatly accelerated.

**Warning:** if the code is strongly limited by memory bandwidth, no SIMD technique can bridge this gap.

# Ideal scenario for applying SIMD to a loop

- All iterations are independent
- There is no branch in the loop body
- The arrays are accessed with a stride of one

Example:

```
for (i=0; i<N; i++)
  r[i] = x[i] + y[i];
```

(We assume here that the memory regions pointed by `r`, `x`, `y` do not overlap—no aliasing)

# An example of applying SIMD

**Pseudocode** of applying SIMD (assuming that each SIMD register can store 4 values):

```
int i, rest = N%4;
for (i=0; i<N-rest; i+=4) {
  load R1 = [x[i],x[i+1],x[i+2],x[i+3]];
  load R2 = [y[i],y[i+1],y[i+2],y[i+3]];
  R3 = ADD(R1,R2);
  store [r[i],r[i+1],r[i+2],r[i+3]] = R3;
}
for (i=N-rest; i<N; i++)
  r[i] = x[i] + y[i];
```

# Beware of loop dependency!

If a loop iteration depends on the result of another iteration—**loop-carried dependency**

```
for (i=start; i<end; i++)
  A[i] = 10.0*A[i+offset];
```

If `offset`<0 $\rightarrow$ **real** dependency (read-after-write hazard)
If `offset`>0 $\rightarrow$ pseudo dependency (write-after-read harzard)

# When there is loop-carried dependency...

In case of real dependency, SIMD cannot be applied if the negative `offset` size is smaller than the SIMD width. For example,

```
for (i=start; i<end; i++)
  A[i] = 10.0*A[i-1];
```

In case of pseudo dependency, SIMD can be applied. For example when `offset`>0,

```
for (i=start; i<end; i++)
  A[i] = 10.0*A[i+offset];
```

# Risk of aliasing

Is it safe to vectorize the following function?

```
void compute(int start, int stop, double *a, double *b) {
  for (int i=start; i<stop; i++)
    a[i] = 10.0*b[i];
}
```

# Risk of aliasing (cont'd)

A problem of "aliasing" will arise if the `compute` function is called as follows

```
compute(0, N-1, &(array_a[1]), array_a);
```

If a programmer can guarantee that aliasing won't happen, this hint can be provided to the compiler.

# The role of compilers

A compiler translates a program, which is implemented in a programming language, to machine code.

A compiler can carry out code optimization of various degrees, dictated by the compiler options provided by the user. (`-O0, -O1, -O2, ....`)

Different compilers probably allow different compiler options, should refer to the user manual!

Numerical accuracy **may** suffer from too aggressive compiler optimizations.

# Profiling

Profiling—gather information about a program's behavior, especially its use of resourses. The purpose is to pinpoint the "hot spots", and more importantly, to identify any performance optimization opportunities (if any) and/or bugs.

Two apporaches of "information gathering":

- Instrumentation—compiler automatically inserts some code to *log* each function call during the actual execution
- Sampling—the program execution is interrupted at periodic intervals, with information being recorded

# GNU gprof

One well-known profiler: GNU gprof
`https://sourceware.org/binutils/docs/gprof/`

- Step 1: compile and link the program with profiling enabled;
- Step 2: execute the program to generate a profile data file;
- Steo 3: run `gprof` to analyze the profile data.

(There are other profilers, of course.)

# Hardware performance counters

Knowing how much time is spent where is the first step. But what is the actual reason for "a slow code" or by which resource is the performance limited?

Modern processors feature a small number of *performance counters*, which are special on-chip registers that get incremented each time a certain event occurs.

Possible events that can be monitored:

- number of cache line transfers
- number of loads and stores
- number of floating-point operations
- number of branch mispredictions
- number of pipeline stalls
- number of instructions executed